State-Aware Performance Analysis with eXtended Stochastic Probes

Allan Clark and Stephen Gilmore

University of Edinburgh, Scotland

Abstract. We define a mechanism for specifying performance queries which combine instantaneous observations of model states and finite sequences of observations of model activities. We realise these queries by composing the state-aware observers (called *eXtended Stochastic Probes* (XSP)) with a model expressed in a stochastically-timed process algebra. Our work has been conceived in the context of the process algebra PEPA. However the ideas involved are relevant to all timed process algebras with an underlying discrete-state representation such as a continuous-time Markov chain.

1 Introduction

When modelling complex systems we generally wish to make queries and therefore must describe the set of states in which we are interested. The analysis in question may be a steady-state query asking a question such as: "In the long-run what percentage of its time does the server spend idle?" The set of states in which we are interested is then the *steady-set*. Passage-time queries are often concerned with events, however the query must still be specified as a set of states, which we will call the *passage-set*. To perform a passage-time analysis the solver can extract the set of source states and the set of target states from the passage-set. The set of source states is taken to be all of those states in the passage-set which are the target of some transition whose source lies outside the passage-set. Conversely the set of target states is taken to be the set of states outside the passage-set which are the target of some transition whose source lies in the passage-set.

More generally whether we are performing a steady-state or passage-time analysis we will be interested in specifying the *query-set*. There are currently two kinds of mechanism for specifying query-sets: *state-specifications* and *activity-specifications*.

We are interested in the robustness and portability of our query specifications. For robustness we would like to ensure that our query specification remains correct whenever we make unrelated changes to our model. For portability we would like one query specification to be used over several differing models. Additionally it is important that we are able to make many different queries without needing to alter the model. We have found that the above two query specification techniques alone are insufficient for our aims. Additionally allowing the user to specify their queries using either is still not sufficiently expressive. We have found it necessary to combine the two into one specification language which allows state-specifications to be intermixed within an activity probe specification. This language we have called *eXtended Stochastic Probes* (XSP).

N. Thomas and C. Juiz (Eds.): EPEW 2008, LNCS 5261, pp. 125–140, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Structure of this paper. The rest of this paper is structured as follows: section 2 discusses related work and section 3 formally introduces the two separate specification techniques; state specifications and activity probes as well as formally introducing our extension which allows the combination of both approaches. Section 4 provides details of the conversion of extended probe specifications into our target language PEPA as implemented in our PEPA compiler. A detailed example is provided in sections 5 and 6. Finally conclusions are presented in section 7.

2 Related Work

The use of a regular expression-like language to describe a probe component which is automatically added to a PEPA model was studied by Katwala, Bradley and Dingle [1]. The addition of probe components has been a feature of the Imperial PEPA Compiler [2] (now the International PEPA Compiler) since it was first developed and remains so in the derivative work ipclib [3].

Stochastic probes describe activity-observations. We have previously extended this formalism to locate activities within structured models [4]. We introduced immediate actions into communicating local probes to convey state information without perturbing the performance analysis which was being made. In the present work we add state-observations to the existing stochastic probes language which specifies location-aware activity-observations.

A widely-used language for describing logical properties of continuous-time Markov chains is CSL (Continuous Stochastic Logic), introduced by Aziz, Sanwal, Singhal and Brayton [5]. An application of CSL to a process algebra must first translate the higher-level state information exposed to the user to the states of the Markov chain. The well-formed formulae of CSL are made up of *state formulae* ϕ and *path formulae* ψ. The syntax of CSL is below.

$$\begin{aligned}\phi &::= \mathit{true} \mid \mathit{false} \mid a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \\ &\qquad \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{S}_{\bowtie p}[\phi] \\ \psi &::= \mathrm{X}\,\phi \mid \phi\ \mathrm{U}^{I}\ \phi \mid \phi\ \mathrm{U}\ \phi\end{aligned}$$

Here a is an atomic proposition, $\bowtie \in \{<, \leq, >, \geq\}$ is a relational parameter, $p \in [0, 1]$ is a probability, and I is an interval of $\mathbb{R}$. Derived logical operators such as implication ($\Rightarrow$) can be encoded in the usual way.

The implementation of the CSL logic in the model-checker PRISM [6] is extended with additional state-specifications called *filters*. An example is shown in the following formula where

$$\mathcal{P}_{>0.97}[\mathit{true}\ \mathrm{U}^{I}\ \phi_2\{\phi_1\}]$$

determines whether the probability of, from a state satisfying the filter ϕ_1, reaching a state satisfying ϕ_2 within interval I is greater than 0.97.

Because CSL can only describe states and not the events which cause state transitions – namely actions – CSL (without filters) was extended with activity-specifications by Baier, Cloth, Haverkort, Kuntz and Siegle [7] to provide the language asCSL. In that work the authors added activity observations to a state-aware logic. This has recently been further extended to provide the language CSL^{TA} [8] by Donatelli, Haddad

and Sproston. This language allows properties referring to the probability of a finite sequence of timed activities through the use of a timed automaton.

The XSP language presented here is close to the language which would be obtained by extending asCSL with the state-filters used in PRISM. However it would extend the language of asCSL+filters with observations of activities at locations within a hierarchically-structured performance model. All prior activity-aware variants of CSL (including asCSL and CSL^{TA}) make observations of a Markov chain model without hierarchical component structure. This entails that they cannot be used (say) to distinguish arrivals to server 1 from arrivals to server 2 in the example below

$$Client \underset{\{arrive\}}{\bowtie} \Big((Server1 \underset{\mathcal{L}}{\bowtie} Network) \parallel (Server2 \underset{\mathcal{M}}{\bowtie} Raid) \Big)$$

but this distinction can be expressed in the language XSP.

3 State and Probe Specifications

The models which we consider consist of compositions of multiple copies of sequential components cooperating on shared activities. A state-specification is a predicate involving expressions over the multiplicities of the sequential components in a system. The expressions in the predicate may compare a multiplicity to a constant or to the multiplicity of another component. Typical predicates test for the presence, absence or abundance of a particular component but more complex arrangements are possible. For example $ClientWait > 2 \times ServerReady$ specifies those states in which the number of clients waiting is more than twice the number of ready servers. The full syntax for state-specification equations is given in Figure 1(left).

An activity-specification is a labelled regular expression describing the sequence of activities which lead into and out of the query-set. The labels start and stop are used to indicate the activities which enter and exit the query-set respectively. Activity-specifications are realised as *stochastic probes* which are automatically translated into a component which is then attached to the model.

Probes may be attached *globally* to the entire model (thereby observing all of the model behaviour) or *locally* to a specific component (therefore observing from the perspective of this component). The probe cooperates with the component to which it is attached over all of the activities in its alphabet. It is important that the probe is always willing to perform all of these activities in each of its local states in order that it does not alter the behaviour of the model.

A very simple probe may specify the set of states between a *begin* and an *end* activity: *begin*:start, *end*:stop. More complex queries are possible such as:

$$((pass, pass, pass)/send)\text{:start}, send\text{:stop}$$

This specifies that if we observe three *pass* activities without observing a *send* activity then the model has entered the query-set. When a *send* activity has been observed then the model has left the query-set. The full syntax for activity-probe specifications is given in Figure 1(right).

name	:=	*ident*	process name
pred	:=	$\neg$*pred*	not
	|	true | false	boolean
	|	if *pred* then *pred* else *pred*	conditional
	|	*pred* $\vee$ *pred*	disjunction
	|	*pred* $\wedge$ *pred*	conjunction
	|	*expr*	expression
expr	:=	*name*	multiplicity
	|	*int*	constant
	|	*expr relop expr*	comparison
	|	*expr binop expr*	arithmetic
relop	:=	$= \mid \neq \mid > \mid <$	
	|	$\geq \mid \leq$	relational operators
binop	:=	$+ \mid - \mid \times \mid \div$	binary operators

P_{def}	:=	*name* :: R	locally attached probe
	|	R	globally attached probe
R	:=	*activity*	observe action
	|	R_1, R_2	sequence
	|	$R_1 \mid R_2$	choice
	|	R:label	labelled
	|	$R\ n$	iterate
	|	$R\{m,n\}$	iterate
	|	R^+	one or more
	|	R^*	zero or more
	|	$R^?$	zero or one
	|	$R/activity$	resetting
	|	(R)	bracketed

$$R := \ldots \mid \{pred\}R \text{ guarded}$$

Fig. 1. The grammar on the left defines the syntax for state-specifications while the grammar on the right defines the syntax for activity-probe-specifications. The grammar extension at the bottom defines the additional syntax for eXtended Probe Specifications.

Activity probes have two abstract states, *running* and *stopped*. An *abstract* state of a (component of a) model, is a set of states with a common property. When the probe is in between the two labels start and stop the probe is said to be in the running state and otherwise in the stopped state.

Probes are stateful components which advance to a successor state whenever an activity is observed which is in the *first-set* of the probe. The first-set of a probe is the set of activities which are enabled at the current position of the probe specification. For example the probe $(a|b), R$ can advance to a state represented by the probe R on observing the activities a or b and its first-set is $\{a, b\}$. A given probe will *self-loop* on any activity which is in the alphabet of the full probe but is not in the current first-set. This means that the probe observes the occurrence of the activity and hence does not prevent the model from performing it, but does not advance.

The novelty in the present paper is the combination of state-specifications, activity-specifications and both local and global observations.

We do this by allowing a sub-probe of an activity-probe specification to be *guarded* by a state-specification. Having already done the work of describing states earlier, the additional syntax—shown in Figure 1(bottom)—is very light.

The meaning of the probe $\{p\}R$ is that any activity which begins the probe R must occur when the state of the model satisfies the state-specification predicate p. If this predicate is not satisfied then the probe self-loops on the given activity. For example the extended probe: $\{Server_broken > 0\}request$: start, $response$: stop is similar to a common query which analyses the response time between the two activities *request* and *response*. Here though the initial observation of the *request* activity is guarded by the state specification $Server_broken > 0$ and hence all occurences of *request* will be

ignored by this probe unless there is at least one process in the *Server_broken* state. This could be used to analyse the response time in the specific case that at least one server is down.

In this paper we will express models in the stochastic process algebra PEPA [9] extended with *functional rates* [10] (known as "marking-dependent rates" in Petri nets). The definition $P \stackrel{def}{=} (\alpha, f).P'$ denotes a component P which performs an activity α at an exponentially-distributed rate determined by evaluating the function f in the current model state. After completing activity α, P behaves as P'.

Another model component $Q \stackrel{def}{=} (\alpha, \top).Q'$ could *cooperate* with P on activity α thus:

$$P \underset{\{\alpha\}}{\bowtie} Q$$

We write $P \parallel Q$ when the cooperation set is empty and $P[3]$ as an abbreviation for $P \parallel P \parallel P$. The component $R \stackrel{def}{=} (\alpha, r_\alpha).R' + (\beta, r_\beta).R''$ chooses to perform activity α with rate r_α with probability $r_\alpha/(r_\alpha + r_\beta)$, and β similarly. The process $a.P$ performs an immediate action a and evolves to P.

4 Implementation

The XSP language is implemented in the International PEPA Compiler (ipc), a stand-alone modelling tool for steady-state and transient analysis of PEPA models. When presented with a PEPA model and an XSP probe, ipc first translates the probe specification into a PEPA component. This translated component is then attached to the model to form a new model. It is this subsequent model which ipc then translates into a Markov chain representing the augmented model and solves the resulting Markov chain for the stationary probability distribution. In the case of a passage-time analysis uniformisation [11,12] is then used to compute the probability density and cumulative distribution functions for the passage across the XSP probe.

Translating probe specifications into valid PEPA components and attaching them to the model before any compilation of the model is performed has several advantages. The user may provide several probe specifications which are translated and added to the model in turn resulting in subsequent augmented models. Thus additional probes may refer not only to activities (and immediate actions) performed by the original model but also those performed by other probes. In this way probes may use immediate actions to perform immediate communication between probe components. Furthermore, although in this paper we have focussed on translating the model augmented with the translated probe using ipc via its Markov Chain representation, we could also analyse the augmented model using other techniques developed for analysing PEPA models, notably stochastic simulation and translation to ordinary differential equations[13] allowing us to cope with models with much larger state spaces. Finally the static analysis used to reject (or warn about) suspect PEPA models can now be run over the entire augmented model including the translated probe components providing further assurance that we have not made a mistake with our specification. This is in addition to some sanity checking over the probe specification itself.

The implementation of XSP follows a tradition of translating regular-expression languages to finite-state automata. We first translate the probe specification into a non-deterministic finite-state automaton. This cannot itself be translated directly into a PEPA component so we next translate this into a deterministic finite automaton. Having done this the self-loops may then be added to each state (recall from section 3 that self-loops must be added to each state of the probe to avoid the probe affecting the behaviour of the model).

Although for some probe specifications it is unavoidable that we increase the state space of the model we wish to keep the cost of this as low as possible. With this in mind the translated deterministic finite automata is minimised. It is the minimised DFA with the addition of the self-loops which can be translated directly into a PEPA model. This final step is a trivial re-formatting stage – we must only take into account whether or not the model performs each observed action as a timed activity or an immediate action. In the case of the former the probe component must passively observe the activity at rate $\top$ and in the case of the latter it is simply added itself as an immediate action.

Probe definitions, and in particular local probe definitions, may use labels to communicate important events to a master probe which the user provides. The :start and :stop labels are special cases of this communication whereby the event is the transition of the probe into the abstract running or stopped states. All communication labels are implemented as immediate actions so as not to distort the behaviour of the model. Care must be taken not to add self-loops to a state in which immediate communication is possible in case the observed action on which the self-loop is performed is itself immediate, which would lead to non-determinism.

The guards on the activities of a probe in an extended probe specification are implemented as guards on the activities of the translated probe component. These in turn may be implemented as functional rates in which the rate is zero if the predicate is false. Care must be taken when adding the self-loops. Previously a self-loop on activity x in the alphabet of the probe was added to a given state if activity x could not currently be performed to advance the state of the probe. Now whenever it is possible for a guarded activity x to advance the state of the probe we must add a self-loop for the case in which the guard is false. However it must not self-loop whenever the guard is true hence the self-loop is itself guarded by the negation of the guard predicate.

To attach the translated probe component to the model we synchronise over the alphabet of the probe. For a global probe it is trivial to attach since we cooperate with the whole model. For the global probe, if *Probe* is the name given to the translated PEPA component in the initial state of the probe and *System* is the original system equation then the augmented model's system equation is given by:

$$\mathit{Probe} \underset{\mathcal{L}}{\bowtie} \mathit{System}$$

where $\mathcal{L}$ is the alphabet of the probe. A local probe $P :: R$ is attached by descending through the cooperations (and hiding operators) which make up the *System* component. We attach the probe to the leftmost occurrence of P splitting an array if required. Therefore if *System* is represented by the cooperation $(L \underset{\mathcal{M}}{\bowtie} P[4]) \underset{\mathcal{N}}{\bowtie} Q$ then the system equation of our augmented model becomes:

$$(L \underset{\mathcal{M}}{\bowtie} ((\mathit{Probe} \underset{L}{\bowtie} P) \parallel P[3])) \underset{\mathcal{N}}{\bowtie} Q$$

5 An Example Scenario

Our example scenario involves the arbitration of many processes accessing a shared resource. Here we are considering a symmetric multi-processing architecture in which there are several processors which must be allowed access to a shared memory. However the models and query specifications can be applied to similar scenarios involving access by many clients to a shared resource, for example a wireless network in which the clients must compete to send or receive over a shared channel.

With our models we wish to compare choices for arbitration. Here we will compare a round-robin scheme with a first-come, first-served queueing system. In the round-robin scheme each client is given the chance to use the shared resource in turn, at each such turn the client may choose to pass up the opportunity or it may use the resource. In a first-come, first-served queue a client continues to work without the shared resource until it is required and then signals its interest in access to the shared resource. At this point the client is put to the end of the queue of clients and must wait until all the clients ahead of it in the queue have finished with their turn at the resource before being granted access.

We will be concerned with the time it takes from after a specific client has performed some internal *work* (indicating that it is now ready to use the shared resource) until after it has completed a *send*. Here the send activity is used as the name for accessing the shared resource and can be thought of as either sending data to the shared memory in a symmetric multi-processor environment or using the shared channel to send data in a wireless network.

5.1 The Round-Robin Model

For the round-robin scheme we model the resource as a token which may be in one of several places where each place represents a slot in which exactly one client may use the resource. A client is able to perform the *work* activity before being able to use the resource. It must cooperate with the resource and can of course only do this if the token of the resource is in the correct place. In addition to being able to perform a *work* activity the client may pass up the opportunity to use its slot. The *Client* component then is modelled as:

$$\begin{aligned} Client &\stackrel{def}{=} (work, work_rate).Wait \\ &\quad + (pass, \top).Client \\ Wait &\stackrel{def}{=} (send, send_rate).Client \end{aligned}$$

The resource is modelled by the *Token* process. The *Token* when in position zero may cooperate with the client over the *send* or the *pass* activity. To model each of the other places for the token we could model more clients. Instead we assume that the token moves on from each place at a given rate which encompasses both the possibilities that the respective client sends or passes. In position i the token is modelled by:

$$Token_i \stackrel{def}{=} (delay, delay_rate).Token_{i-1}$$

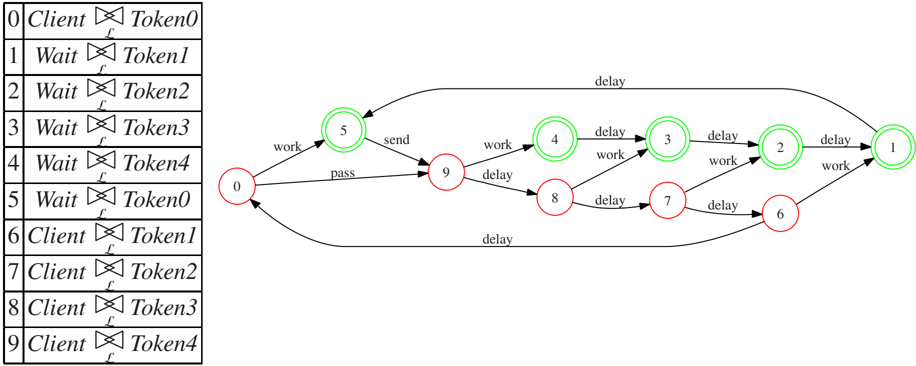
When the token is in position zero it is defined as:

$$\begin{aligned} Token_0 &\stackrel{def}{=} (send, \top).Token_M \\ &\quad + (pass, pass_rate).Token_M \end{aligned}$$

where M is the number of other places/clients on the network. The main system equation is defined to be:

$Client \underset{\mathcal{L}}{\bowtie} Token0$ where $\mathcal{L} = \{send, pass\}$

Figure 2 depicts the entire state space of the model where M is set to four.



0	$Client \underset{\mathcal{L}}{\bowtie} Token0$
1	$Wait \underset{\mathcal{L}}{\bowtie} Token1$
2	$Wait \underset{\mathcal{L}}{\bowtie} Token2$
3	$Wait \underset{\mathcal{L}}{\bowtie} Token3$
4	$Wait \underset{\mathcal{L}}{\bowtie} Token4$
5	$Wait \underset{\mathcal{L}}{\bowtie} Token0$
6	$Client \underset{\mathcal{L}}{\bowtie} Token1$
7	$Client \underset{\mathcal{L}}{\bowtie} Token2$
8	$Client \underset{\mathcal{L}}{\bowtie} Token3$
9	$Client \underset{\mathcal{L}}{\bowtie} Token4$

Fig. 2. States of the round-robin model with a passage-time analysis states marked

5.2 The Queue Model

The queue model is a little more complex since the client we are analysing may join the queue at any time but must only be served when it is at the head of the queue. The queue is modelled in a similar fashion to the *Token* process. It may be in one of M states $Queue_i$ where i is the current length of the queue.

The client is now modelled as being in a state of working or in one of a set of M states $Client_i$ each of which corresponds to a position in the queue. When the client performs the *work* activity and is ready to use the shared resource it cooperates with the *Queue* process over an action which indicates into which state the client should proceed. Only once the client is in state $Client_0$ can it perform the *send* activity which will end our passage of interest. Again the other clients in the model may be modelled explicitly but here we allow the queue to move from state $Queue_i$ to state $Queue_{i+1}$ at the (functional) rate $(M - i - Client) \times work_rate$ since when there are i clients in the queue there will be $M - i$ clients which may join the queue. We subtract one from that if the queried client is not in the queue since this performs its own *work* activity to join the queue. The full model is shown in the appendix.

5.3 The Random Model

The random model is used for comparison. The random scheme operates in a similar fashion to the queue scheme, except that there are a number of clients in the queue and the client which is given access to the shared resource is entirely random. It may be the client that was the first to enter the queue but it may be the client that was last to enter the queue.

In this model we do model the other clients. The client is defined as for the round-robin model except that it need not perform a *pass* activity.

$$\begin{aligned} Client &\stackrel{def}{=} (work, work_rate).Wait \\ Wait &\stackrel{def}{=} (send, \top).Client \end{aligned}$$

The queue as before may be in one of i states representing how many clients are in the queue. The queue now cooperates with a random waiting client to perform the *send* activity or a random working client to perform a *work* activity. The queue with i waiting clients is defined as:

$$\begin{aligned} Queue_i &\stackrel{def}{=} (work, \top).Queue_{i+1} \\ &+ (send, send_rate).Queue_{i-1} \end{aligned}$$

In position zero the queue cannot perform a *send* activity and cannot perform a *work* activity when the queue is full. Finally the system definition is given by:

$$Client[5] \underset{\{work,send\}}{\bowtie} Queue0$$

5.4 The Passage-Time Analysis

With these models we wish to analyse the expected time it takes for the resource to be granted to the client once the client is ready. For this we wish to analyse from after a *work* activity has been performed until after a *send* activity has been performed. We therefore must identify the passage-set. That is, the set of states which lie between those two events.

In Figure 2 the states in the passage-set for this particular query are identified using double circles.

To specify this set using a state-specification we must use our knowledge of the system to identify the conditions which hold at all of the states in the passage-set.

For the round-robin model this is simply when the client is in the *Wait* state.

$Wait = 1$

A similar specification also works for the random model with the caveat that we must specify which *Client* we consider. For the queue model it is whenever the client is in any of the queue states.

$Client1 = 1 \vee Client2 = 1 \vee Client3 = 1 \vee Client4 = 1 \vee Client5 = 1$

To specify this query using an activity probe we use the two activities themselves as the begin and end events for the probe. The probe definition is given as:

$Client :: work{:}\mathsf{start}, send{:}\mathsf{stop}$

Note that this same probe works for all three models. For the round-robin and queue models it is not strictly necessary for us to attach the probe to the *Client* component since there is only one client component which may perform the observed activities. However doing so leads to a more robust probe as evidenced by the fact that the same probe can be used for the random model in which there are additional client processes.

Having performed this analysis for all three models we can compare the speed with which each arbitration method allows a waiting client to use the shared resource. Figure 3 shows a comparison of both the cumulative distribution function and the probability density function for the passage-time queries on the three models representing the three arbitration schemes. These functions have been evaluated by applying the uniformisation procedure [11,12] to the CTMC which is generated from the PEPA model.

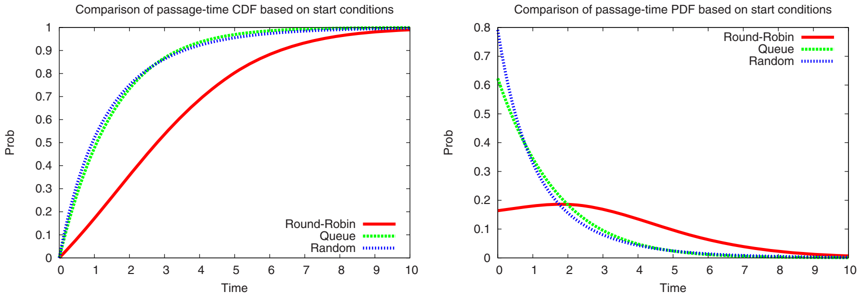


Fig. 3. Comparison between the passage-time results for the three models

From the results we can see that the queue and random models perform very similarly and both outperform that of the round-robin scheme.

The robustness of the query specification in general depends on what the modeller is likely to modify. In our example above the state-specification is vulnerable to any change in the model which increases the number of states in which the client may be in either the abstract state of 'waiting' or the abstract state of 'working'. The abstract state of 'waiting' in our model corresponds to exactly one state of the client, namely: *Wait*. Similarly the abstract state of 'working', which is used to specify that the *send* activity has completed, maps to exactly one component state, namely: *Client*. If the model is modified such that either of these two mappings from abstract state to a concrete set of states is disturbed then the state specification will be invalid and must be revised.

In contrast the activity probe need not be modified since there could for example be any number of unobserved activities and associated intermediate states between the *work* and the *send* activities. However if we modify the set of activities which may cause the model to transition between the abstract states then we must revise our probe specification. For example above there were only two activities which the *Token*0 component may perform to become a *Token*4 component, namely : *pass* and *send*. However if this were to change then our probe specification would be invalid and would require updating.

5.5 Splitting the Analysis

We may wish to partition the passage-time results we have obtained for our three models to enable us to report the expected time the client has to wait depending on the state of the model at the time at which the client becomes ready to use the shared resource. So for example in the round-robin model above we may wish to ask the question: "What is the expected time between the client performing a *work* activity and the client performing a *send* activity given that the *work* activity occurs when the token process is in state *Token*4?" This question may be of particular interest because it represents the worst case scenario. We have shown that the overall performance in the general case of the round-robin scheme is worse than that of the queue and random schemes. However it may be that the round-robin scheme has less erratic performance in that it matters less at what time the client becomes ready to use the shared resource. It may be that

the worst case performance for the round-robin scheme is better than that for both the queue and random schemes. This may be of particular interest in say a network, where traffic can become congested at particular times and hence the worst case performance is of more interest than the average case performance.

To write this exact query as a state-specification we must resort to specifying the source set and the target set explicitly. This is because if we specify the states as a passage-set it will include the states where the token is in places $1 \dots 3$, while the client is still waiting. Clearly these states are reachable by a transition from a state outside the passage-set. In fact specifying the passage-set in this manner would give identical results to analysing the time the client must wait regardless of when the *work* activity was completed. With this in mind our source and target sets for the round-robin model worst case scenario are specified respectively by:

$$source : Wait = 1 \wedge Token4 = 1$$
$$target : Client = 1$$

Note however that it is a little unsatisfactory that we had to know so much about the behaviour of the model. Even if one considers this a good thing – modellers should know about the behaviour of their models – the query specification is very fragile in that if we modify our model it is likely that this query specification must also be updated. In addition the target set is larger than necessary. This will not affect the results of the analysis but may cause the analysis time to increase. Again a very similar state-specification can be used for the random model.

For the queue model worst case scenario analysis we can use our knowledge of the system to make our state specification simpler than in the average case, this is because there are fewer source states. Our state query is written as:

$$source : Client5 = 1$$
$$target : Client = 1$$

To write this query as an activity probe we must identify a sequence of activities which will place the model in the source-set and the sequence of activities which will complete the passage (from a source state). Specifying this using an activity probe means that the query need not be split up. This is because the probe is in the abstract running state only when it has passed through a source state. This means that we need not split our specification into two separate ones however the drawback is that the state space is increased. Our query for the round-robin model is specified by:

$$Client :: ((pass|send), work)/delay{:}\mathsf{start}, send{:}\mathsf{stop}$$

The $(pass|send)$ component ensures that the token has moved to state *Token*4 before we observe the *work* activity. By restricting the *delay* activity (with $/delay$) we assert that the probe will not move past the start label unless the sequence ending with the *work* occurrence does not contain a *delay* activity. This in turn ensures that the token is in the state *Token*4 when the probe transitions to running. If a *delay* is observed this resets the probe which must then wait to observe a *pass* or *send* once again.

The state space is increased because there are states in the passage which must be duplicated. For example the state in which the token is in state *Token*3 and the client is in state *Wait* is duplicated since the probe component may be in either the running or the stopped state depending on whether the given state was reached via a source-state.

For both the random model and the queue model specifying this condition as an activity probe is particularly difficult. In the following section we detail a far more portable and robust method of obtaining these analysis results, namely the use of eXtended Stochastic Probes (the XSP language).

6 Using eXtended Stochastic Probes

In the previous section we discussed the two main methods for specifying a query-set. Both have advantages and disadvantages and can be used in different circumstances. We used these to obtain a passage-time analysis and then proceeded to split this into distinct queries depending on the state of the model when the passage is begun. We have shown that either of the two methods alone are sometimes unsatisfactory. In this section we provide the same split queries using the combined approach, eXtended Stochastic Probes. The following probe can be used on the round-robin model to analyse the passage in the worst case when the token is as far away as possible.

$$Client :: \{Token4 = 1\}work\text{:start}, send\text{:stop}$$

This probe will only be started by an observation of the *Client* performing a *work* activity if the token is currently in the state *Token*4. All other occurrences of the *work* activity will be ignored.

To specify the same worst case scenario query for the queue we can specify the extended probe:

$$Client :: \{Queue4 = 1\}work\text{:start}, send\text{:stop}$$

This specification works in exactly the same way. The only difference is the name of the state of the resource in the worst case scenario. For the random model the probe is exactly the same.

Without changing the models we can make additional queries corresponding to all of the different possible states of the resource at the time at which the client becomes ready to make use of the shared resource. In the case of the round-robin scheme this is the different places that the token may be in. In the case of queue and random models this is the length of the queue. We provide the probes:

$$Client :: \{TokenN = 1\}work\text{:start}, send\text{:stop}$$
$$Client :: \{QueueN = 1\}work\text{:start}, send\text{:stop}$$

The graphs in Figure 4 show the cumulative distribution and probability density functions of the passage-time responses given by restricting the probe to the conditions of the shared resource.

6.1 Discussion of Results

With the basic analyses we determined that the average case response-time was worst for the round-robin scheme and very similar for the queue and random schemes. The results for the individual circumstances for the round-robin and the queue models are identical. This is because in both cases the number and rates of the timed activities that we must observe between the source and target are identical for each equivalent circumstance. For example when the queue is empty and the token is in the correct place both models are only measuring one activity, namely the *send* activity. This tells

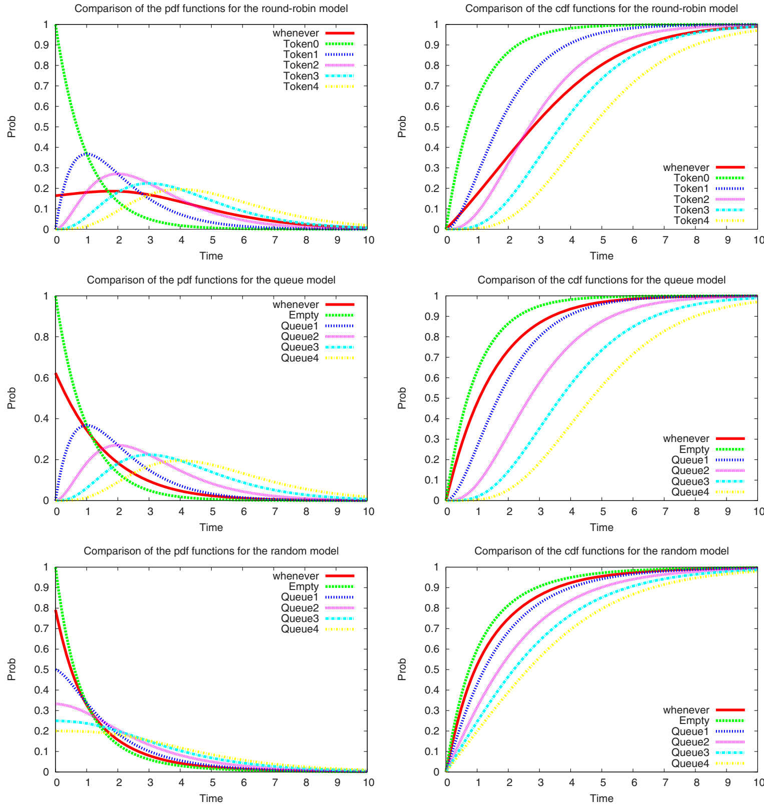


Fig. 4. Graphs of the PDF and CDF functions for the split passage-time results for the three models

us that the reason the average case is worse for the round-robin model is because the unfavourable cases happen more frequently than for the queue model. This may only be the case because of the particular rate values which we have chosen and we may wish to change those rates to see if we could make the round-robin model outperform the queue model (in the average case). Indeed one should perform these experiments with varying rates for both models as we have previously done using a distributed computing platform to analyse the separate instances of each model [14].

In the case of the random model we can see that although the average case performance is very similar to that of the first-come, first-served queue the performance is actually much less varied. This is because for each state of the queue at the time of the client becoming ready to use the resource there are still more paths to the target states over which to average out the performance. For example if the client becomes ready when the queue is empty this is no guarantee that our client will be the next client to use the resource. Similarly if the queue is full we may still be the next client to use the resource. The random queue may have some other less desirable properties, for example a client may wait in the queue while arbitrarily many other clients are processed ahead

of it. However our results show that – at least for the parameters we have specified – a client is highly unlikely to spend a long time in the queue.

7 Conclusions

We have described an extension – eXtended Stochastic Probes – to the language of stochastic probes. Our extension allows the modeller to refer to the states of components which are located in a hierarchically-structured performance model expressed in the stochastic process algebra PEPA.

We consider state-specifications alone to be insufficient since they cannot be used to distinguish states based on the activities which have been performed to reach that state. Sometimes to perform the desired analysis we must increase the state-space of the model and state-specifications offer no way to do this automatically. Activity probe specifications are also insufficient for all purposes and in particular are poor at describing states which represent a balancing of activities. This is a frequent kind of query such as "how likely is the server to be operational?" which may be the result of a balance of 'break' and 'repair' activities. Finally allowing *either* state specifications *or* activity probes is still not an acceptable solution. Situations which call for a combination of the two approaches arise when the modeller wishes to combine observations with state descriptions. A common example of such a combination is to ask about the response time when the request is made at a time when a particular system component is in a particular (abstract) state. A standard query is: "What is the response time when at least one of the servers is broken".

We have shown an example consisting of three models describing similar scenarios but each using different modelling techniques. In the round-robin and queue models we have represented only the client component that we wish to analyse while in the random model all of the clients in the system were represented explicitly. The queue model makes use of immediate actions and functional rates. Despite this the extended probe specifications we used to split-up our passage-time analyses were portable across the three models.

Our language of extended probe specifications has been fully implemented in the ipclib library used and distributed with the International PEPA Compiler. This is available for download as open source software from `http://www.dcs.ed.ac.uk/pepa/tools/ipc`.

Acknowledgements. The authors are supported by the EU FET-IST Global Computing 2 project SENSORIA ("Software Engineering for Service-Oriented Overlay Computers" (IST-3-016004-IP-09)).

References

1. Argent-Katwala, A., Bradley, J., Dingle, N.: Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models. In: Proceedings of the Fourth International Workshop on Software and Performance, Redwood Shores, California, USA, pp. 49–58. ACM Press, New York (2004)

2. Bradley, J., Dingle, N., Gilmore, S., Knottenbelt, W.: Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. In: Kotsis, G. (ed.) Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, University of Central Florida, pp. 344–351. IEEE Computer Society Press, Los Alamitos (2003)
3. Clark, A.: The ipclib PEPA Library. In: Harchol-Balter, M., Kwiatkowska, M., Telek, M. (eds.) Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), pp. 55–56. IEEE, Los Alamitos (2007)
4. Argent-Katwala, A., Bradley, J., Clark, A., Gilmore, S.: Location-aware quality of service measurements for service-level agreements. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 222–239. Springer, Heidelberg (2008)
5. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. ACM Trans. Comput. Logic 1, 162–170 (2000)
6. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
7. Baier, C., Cloth, L., Haverkort, B., Kuntz, M., Siegle, M.: Model checking action- and state-labelled Markov chains. In: DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, Washington, DC, USA, p. 701. IEEE Computer Society, Los Alamitos (2004)
8. Donatelli, S., Haddad, S., Sproston, J.: CSL^{TA}: an Expressive Logic for Continuous-Time Markov Chains. In: QEST 2007: Proceedings of the Fourth Interational Conference on Quantitative Evaluation of Systems, Washington, DC, USA, pp. 31–40. IEEE Computer Society, Los Alamitos (2007)
9. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
10. Hillston, J., Kloul, L.: An efficient Kronecker representation for PEPA models. In: de Alfaro, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 120–135. Springer, Heidelberg (2001)
11. Grassmann, W.: Transient solutions in Markovian queueing systems. Computers and Operations Research 4, 47–53 (1977)
12. Gross, D., Miller, D.: The randomization technique as a modelling tool and solution procedure for transient Markov processes. Operations Research 32, 343–361 (1984)
13. Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, pp. 33–43. IEEE Computer Society Press, Los Alamitos (2005)
14. Clark, A., Gilmore, S.: Evaluating quality of service for service level agreements. In: Brim, L., Leucker, M. (eds.) Proceedings of the 11th International Workshop on Formal Methods for Industrial Critical Systems, Bonn, Germany, pp. 172–185 (2006)

A. The Full Queue Model

A.1. The Client Behaviour

This component represents the system workload.

$$
\begin{aligned}
Client &\stackrel{def}{=} (work, work_rate).ClientQ \\
&+ (delay, \top).Client \\
ClientQ &\stackrel{def}{=} place0.Client1 + place1.Client2 + place2.Client3 \\
&+ place3.Client4 + place4.Client5 \\
Client1 &\stackrel{def}{=} (delay, \top).send.Client \\
Client2 &\stackrel{def}{=} (delay, \top).Client1 \\
Client3 &\stackrel{def}{=} (delay, \top).Client2 \\
Client4 &\stackrel{def}{=} (delay, \top).Client3 \\
Client5 &\stackrel{def}{=} (delay, \top).Client4
\end{aligned}
$$

A.2. The Queue Component

This model component has the responsibility of correctly implementing the intended first-in first-out behaviour of the queue. It ensures that the functional rates are correctly evaluated by counting the number (either 0 or 1) of components in the *Client* state.

$$
\begin{aligned}
Queue0 &\stackrel{def}{=} (join, work_rate \times 4).Queue1 \\
&+ place0.Queue1 \\
Queue1 &\stackrel{def}{=} (join, work_rate \times (4 - Client)).Queue2 \\
&+ place1.Queue2 \\
&+ (delay, send_rate).Queue0 \\
Queue2 &\stackrel{def}{=} (join, work_rate \times (3 - Client)).Queue3 \\
&+ place2.Queue3 \\
&+ (delay, send_rate).Queue1 \\
Queue3 &\stackrel{def}{=} (join, work_rate \times (2 - Client)).Queue4 \\
&+ place3.Queue4 \\
&+ (delay, send_rate).Queue2 \\
Queue4 &\stackrel{def}{=} (join, work_rate \times (1 - Client)).Queue5 \\
&+ place4.Queue5 \\
&+ (delay, send_rate).Queue3 \\
Queue5 &\stackrel{def}{=} (delay, send_rate).Queue4
\end{aligned}
$$

A.3. The System Equation

Finally, the model components are composed and required to cooperate over the activities in the cooperation set.

$$Client \underset{\mathcal{L}}{\bowtie} Queue0$$

where $\mathcal{L} = \{delay, place\{0..4\}\}$