

Transformations in PEPA Models and Stochastic Probe Placement

Allan Clark* Stephen Gilmore†

Abstract

We describe a language of transformations over performance models written in PEPA [1]. This language allows many similar models to be derived from a single input model. This in turn allows experimentation over a variety of similar configurations of a modelled system. We then turn our attention to the use of transformations as a probe placement language. A stochastic probe is an observational component which may be attached to the entire system as a global observer or to a sub-portion of the model as a local observer. Local observers allow the calculation of such analyses as the response-time of a service as observed by a single client. We use our transformation language as a way for the user to describe where in the system a local probe or probes should be placed.

1 Introduction

Computer systems are often modelled in order to predict performance characteristics. Often the modeller is interested in variations on the exact system modelled. This may be because the system in question is yet to be built, bought, or in some other way realised, and the modeller wishes to explore options on the final system. It may also be because the modeller wishes to know how the current or future system will react to changes in the environment in which the system is deployed. In general, a thorough investigation of potential performance will be based on a family of related models, not just a single one. Our concern in this paper is with generating this family of related models by applying model transformations to a single source.

For some kinds of queries it is advantageous to re-express the model in a form which simplifies the expression of the query. Such a modification should not be left to the modeller since a mistake will lead them to erroneously analyse a model with non-equivalent behaviour. We use stochastic probes [2, 3] to automatically add probe components to the model. These probe components are generated from a regular-expression like specification language and then automatically attached to the model. Until now the attachment of the probe component to the model was with a very limited choice. Either the probe is attached globally to the entire model and therefore must observe all non-hidden activity occurrences regardless of their origin. Alternatively the probe could be attached to a named component of the system equation although the software would then attach the

*LFCS, University of Edinburgh, a.d.clark@ed.ac.uk

†LFCS, University of Edinburgh, stg@inf.ed.ac.uk

probe to a single copy of the first component encountered with the given name. This restrictive scheme has now been superseded by allowing the user to specify how the generated probe component should be attached to the model using the transformation language detailed herein.

In this paper we work with PEPA, a stochastically-timed process algebra where sequential components are defined using prefix and choice. PEPA models require these sequential components to cooperate on some activities, and hide others. A PEPA model typically consists of several sequential components such as these, placed in cooperation. In the model

$$P \bowtie_{\mathcal{L}} Q$$

the sequential components P and Q cooperate on the activities in the set \mathcal{L} . If activity α is in the set \mathcal{L} then P and Q are required to cooperate on α . If activity β is not in \mathcal{L} then either of P or Q , or both, may perform this activity independently. When \mathcal{L} is empty we write $P \parallel Q$ instead of $P \bowtie_{\emptyset} Q$. We also allow the special cooperation $P \bowtie_{*} Q$ to be a synonym for $P \bowtie_{\mathcal{L}} Q$ where \mathcal{L} is the set of activities performed by both P and Q .

Rates are associated with activities performed by each component. The symbol \top is used to indicate that the component will passively cooperate with another on this activity. In this case the passive component may enable or restrict the activity from being performed by the cooperating component but the rate when enabled is determined by the actively cooperating component. The component $(a, r).P$ performs the activity a at rate r whenever it is not blocked by a cooperating component and becomes the process P . The component $(a, \top).Q$ passively synchronises on a and becomes process Q .

In PEPA models we often work with *arrays* of components. We use arrays to represent workload (such as a number of independent clients) or resources (such as a number of independent servers). We write $P[5]$ to denote five copies of the component P which do not cooperate and $P[5][\alpha]$ to denote five copies of the component P which cooperate on the activity α . That is, $P[5]$ is an abbreviation for $P \parallel P \parallel P \parallel P \parallel P$ and $P[5][\mathcal{L}]$ is an abbreviation for

$$P \bowtie_{\mathcal{L}} P \bowtie_{\mathcal{L}} P \bowtie_{\mathcal{L}} P \bowtie_{\mathcal{L}} P.$$

The PEPA language is formally defined in [1]. Applications of the language are described in [4, 5, 6].

Structure The remainder of this paper is structured as follows: in the following section we review related work, in Section 3 we detail model transformations beginning with those which can be performed through the use of variable rate parameters as well as those which cannot. This allows us to introduce our model transformation language. Section 4 provides a brief overview of the stochastic probe architecture for query specification and describes the use of the model transformation language as a method for the modeller to specify the location within the model at which an observation probe component should be attached. Section 5 gives a full example of the use of our transformation language both to vary the model under inspection and the placement of the observation probe to extract the average response-time of the system concerned. Finally we draw some conclusions in Section 6.

2 Related work

Transformations are often applied to PEPA models to convert them into a form which is easier to analyse because it has a recognisable structure which can be exploited by analysis methods such as those based on product-form decomposition of models. In [7] the authors apply transformations to PEPA models in order to generate new models which have a product-form solution. These newly generated models can be efficiently analysed to deliver results which are an approximation to the true results from the original model. In [8] the authors apply term rewriting to generate variants of a PEPA model systematically. Here the goal is to identify quasi-reversible structure in the model as an intermediate step towards a product-form solution based on earlier work which identified a syntactic characterisation of PEPA models which give rise to quasi-reversible structure [9].

In the above the authors were interested in the discrete Markovian interpretation of a PEPA model but since [10] we also have the possibility to use a continuous fluid-flow interpretation. This has given rise to other opportunities to use model transformation to put a PEPA model in a suitable form to apply the fluid interpretation. In [11] the authors use partial evaluation of a PEPA model to replace a nested composition of components by a single sequential component by partially evaluating the Markovian state-space. The authors of [12] also transform their PEPA models before applying the fluid-flow interpretation. The authors of that paper view passive cooperation as syntactic sugar for a particular active cooperation and their transformation removes uses of the passive rate in PEPA models.

The present work differs from the above in that we are using transformation to generate a related family of models which have many definitions in common, but differ in the model configuration which is specified in the PEPA model's system equation. In this way it is possible for a PEPA modeller to maintain a single PEPA source model and a set of transformations instead of a set of models, with attendant benefits for maintenance and debugging of models. Thus it is perfectly possible to imagine any of the above transformations which help the analysis process being applied to models which we have generated via transformation. The role which we assign to model transformation is familiar practice in model-driven approaches to software development.

3 Model Transformations

Model transformations are a way of deriving a new model from an existing one. This can be done either because we wish to analyse a set of similar models because we are unsure of the actual configuration of the system which we are modelling, perhaps because it is yet to be built or obtained. We also may wish to predict how well an existing system can cope with changes in its operating environment such as an increase in demand. Furthermore we may seek to investigate how a current system can be modified to cope with such changes in operating conditions. So for example we wish to analyse the effect that re-deploying one kind of server as another kind may have on the overall system performance. Redeployment then is a common kind of model transformation.

3.1 Model parameterisation

One frequently-applied change to the configuration of a model is to alter the model parameters. Since our implementation of the PEPA allows identifiers to be used as the initial population size of an array then this is one place where transformations can be conveniently applied. Consider the following model:

$$\begin{aligned} a &= 5 \\ b &= 2 \\ &\dots \text{component definitions} \dots \\ \text{System} &\stackrel{\text{def}}{=} \text{Server}_A[a] \parallel \text{Server}_B[b] \end{aligned}$$

In this case if we wish to analyse the same system with one Server_A component redeployed as a Server_B component we can give the model parameter transformations: $a = 4, b = 3$.

Alternatively if we know we are always going to be re-deploying one to the other we may wish to enforce the constraint that the number of servers always remains the same then we can express the model rate parameters in a different way as shown here:

$$\begin{aligned} \text{total} &= 7 \\ a &= 5 \\ b &= \text{total} - a \end{aligned}$$

In this way we may now re-deploy a server by varying the ‘ a ’ population number parameter and we cannot make the mistake of adding or removing servers. We can still add or remove a server by increasing or decreasing ‘total’ however this will change the number of ‘ b ’ servers indirectly. So in general model parametrisation may be used to reconfigure the system to a degree with the drawback that we initially must set the system up with certain transformations being easier to perform than others.

3.2 A Transformation Language

As an alternative we developed a small language of model transformations with the intent being to automate such redeployment changes. Our language has the basic form: $\text{pattern} \implies \text{replacement}$. A transformation such as this will search the model for a component matching the *pattern* and replace it with the *replacement*. Patterns may explicitly match for a given component, action or expression form, or they may contain *pattern variables* which may be matched against anything in the input model. Pattern variables have a leading question mark. Thus when used in a pattern m and P will match only the PEPA model variables m and P whereas $?m$ and $?P$ are pattern variables which will match any subterm.

The subterm which a pattern variable is matched against is recorded and substituted into the replacement. This is best shown with some simple examples:

Example 1 (*Resizing an array*) *In this transformation we are seeking an array of components named P in the input model and we will substitute this with an array of P components which is three components larger.*

$$P[?m] \implies P[?m + 3]$$

Concretely, if $P[5]$ appears in the input then $P[8]$ will appear in the output and if $Q[5]$ appears in the input then $Q[5]$ will appear in the output.

<i>rule</i>	$:=$	<i>pattern</i> \implies <i>replace</i>	rule
<i>replace</i>	$:=$	<i>pattern</i>	replacement
<i>pattern</i>	$:=$	<i>uppercase</i>	component name
		? <i>uppercase</i>	pattern variable
		<i>pattern</i> <i>cooperationset</i> <i>pattern</i>	cooperation
		<i>pattern</i> [<i>expr</i>][<i>actions</i>]	component array
<i>cooperationset</i>	$:=$		independent
		\boxtimes <small>actions</small>	action list
<i>actions</i>	$:=$? <i>lowername</i>	variable
		<i>lowername</i>	one action
		<i>lowername</i> , <i>actions</i>	many actions
<i>expr</i>	$:=$? <i>lowername</i>	variable
		<i>lowername</i>	rate name
		<i>expr</i> <i>binop</i> <i>expr</i>	binary expression
		<i>expr</i> <i>relop</i> <i>expr</i>	comparison
<i>relop</i>	$:=$	= \neq > <	relational operators
		\geq \leq	
<i>binop</i>	$:=$	+ - \times \div	binary operators

Figure 1: The full grammar of the transformations language

Example 2 (*Increasing a cooperation set*) In this rule we are seeking a cooperation between an array of P components and an array of Q components. When we find this we will add the activity a to the cooperation set, placing an additional requirement on the P components to cooperate with the Q components.

$$P[?m] \underset{\{?s\}}{\boxtimes} Q[?n] \implies P[?m] \underset{\{?s,a\}}{\boxtimes} Q[?n]$$

Example 3 (*Removing an activity from a cooperation set*) In the previous example we were adding the activity a to the cooperation set whereas here we are removing a from the cooperation set. Note that in the second example the cooperation explicitly names the cooperating components as P and Q whereas here we will match any component (including those that are themselves cooperations or arrays of components) provided that they cooperate over a set of activities which includes a .

$$?P \underset{\{a,?s\}}{\boxtimes} ?Q \implies ?P \underset{\{?s\}}{\boxtimes} ?Q$$

Concretely, if $K \underset{\{a,b,c\}}{\boxtimes} L$ appears in the input then $K \underset{\{b,c\}}{\boxtimes} L$ appears in the output.

Example 4 (*Redeploying a component*) In this example we redeploy one P component from a cooperation as a Q component, making the array of P components one shorter and the array of Q components one longer.

$$P[?m] \underset{\{?s\}}{\boxtimes} Q[?n] \implies P[?m-1] \underset{\{?s\}}{\boxtimes} Q[?n+1]$$

The grammar for transformation rules is given in Figure 1.

3.3 Redeployment

Having now seen the transformations language we can re-visit redeployment with the use of transformations.

$$\text{Server}_A[?m] \underset{?l}{\bowtie} \text{Server}_B[?n] \implies \text{Server}_A[?m - 1] \underset{?l}{\bowtie} \text{Server}_B[?n + 1]$$

This has the advantage over the simple use of variable array components that one need not have foreseen this change when the model was authored. In addition we could make sure to apply this change when when Server_A and Server_B are in direct cooperation with each other. In this particular rule we have generalised over the actions within the cooperation set but we could have restricted ourselves to a particular set of activities. If we are applying this rule to many model instances it may be that we wish to restrict when it is applied. Using the variable array syntax does not afford the modeller this expressivity but the transformation language does.

4 Probe Placement

A probe is an observational component which we add to models in order to analyse the behaviour of the model. A probe is a stateful component which observes some of the activities performed by the model through passive cooperation. Upon observing activities the probe component will change state in order to record the occurrence of an activity. The modeller and/or analysing software can then examine the state of the model simply by examining the state of the observation probe. One advantage in doing this is that the same probe may be applied to several different models thereby providing a consistent framework against which to compare those models.

Probes may be written in the host process algebra (in our case PEPA) directly and then attached to the model by hand. Creating the probe itself by hand is error-prone for a variety of reasons in particular adding what are known as the “self-loops”. The self-loops allow the probe component to ignore — that is observe and remain in the same state — activities which it observes in other states. Since the probe component must cooperate over all activities which it at some point observes it must not block the model from performing any activity which does not change its current state. Adding such self-loops is often error-prone. In addition the probe component may be reduced to a minimised state machine which may remove or reduce the extent to which the state-space is increased by the addition of the probe component. For this reason we provide a concise regular-expression-like syntax for specifying stochastic probes which are then automatically translated into PEPA components. The probe specification could also be translated into other process algebra components.

Although a translated probe component could be attached to the model by hand, we prefer an automated method. Often a user will create through some automated means a large number of experiment models which they wish to analyse, for example [13, 14]. The use of probes means that our measurement specification can be portable over all the model instances. The workflow we hope for then is to create a generic model in PEPA and associated portable probe specification in XSP [3]. From the generic PEPA model we generate many specialised PEPA models and from the XSP probe specification we generate

a PEPA component which may be attached to each of the generated models. Since the number of generated models may be in the thousands it is important to have an automated method for attaching probes.

4.1 Local Probes

The simplest way to attach a probe component to the model is to attach it in cooperation with the entire system equation as in:

$$\text{System} \bowtie_{\mathcal{L}} \text{Probe}$$

where \mathcal{L} is the alphabet of the probe — that is the set of activities which the probe component will observe. This places the probe as a global observer, however often this is not appropriate. A common example is when we are calculating the response-time profile of a given service. We are therefore interested in the response-time as observed by each component, not as observed by the service itself. The probe may be specified as:

$$request:start, response:stop$$

If the resulting probe component is attached globally as above then we will incorrectly measure the response-time as observed by the service which will likely be out of sync with that observed by any given client, for more details see [15]. Instead we wish to attach the probe to a single client. Suppose that the System equation is made up by:

$$\text{System} \stackrel{def}{=} \text{Service}[m] \bowtie_{\mathcal{K}} \text{Client}[n]$$

where m is generally less than n and \mathcal{K} is the set of activities which each Client must cooperate with some Service to perform. This cooperation set may be more than simply $\{request, response\}$ for example there may be a *restart*.

In contrast to a global observer probe, we intend for the probe to be attached as a local observer to a single client, thus we transform the system equation into:

$$\text{System} \stackrel{def}{=} \text{Service}[m] \bowtie_{\mathcal{K}} (\text{Client}[n-1] \parallel (\text{Client} \bowtie_{\mathcal{L}} \text{Probe}))$$

The current method for this is to prefix a component name to the front of the probe specification as in:

$$\text{Client} :: request:start, response:stop$$

This tells the software to find a single Client component within the system equation, splitting up an array of such Client components if necessary. For some models this is sufficient, however this has always been a temporary solution knowing that some more sophisticated probe placement language would be required. Consider for example the system in which clients are attached to one or more different kinds of services:

$$(\text{Client}[m] \bowtie_{\mathcal{L}} \text{Service}_A) \parallel (\text{Client}[n] \bowtie_{\mathcal{L}} \text{Service}_B)$$

If we can only prefix probes with component names then there is no way to express that we wish to attach a local probe to a client which is communicating

with Service_B , not one which is communicating with Service_A . Here one could re-write the model however this is not always straightforward and in addition one of our guiding principles in the design of our measurement specification language is never to require the modeller to alter their model. This helps to ensure that our query specifications are robust and reusable.

4.2 Using Transformations

We can use the transformation language as a probe placement language. A probe specification then becomes two things: a probe definition and a transformation rule which describes the context in which the probe is to be used.

The probe name tells the software a name to define the probe component as, this is in order that the user may use this name as part of the replacement part on the right-hand side of a transformation rule. We write the probe definition and give a name to the probe and then couple this with a transformation rule specifying how to place the probe within the model. A simple local probe may be specified as:

- $\text{Probe} = \text{request:start}, \text{response:stop}$
- $\text{Client}[?n] \Longrightarrow (\text{Client} \bowtie_* \text{Probe}) \parallel \text{Client}[?n - 1]$

Notice the use of the special \bowtie_* operator ensuring that the user need not calculate the alphabet of the probe themselves but instead allow the software to do this for them automatically.

Our previous example with two kinds of services in which we were unable to place a local probe onto a client which cooperates with Service_B is now straightforward using the transformation language:

$$\text{Client}[?s] \bowtie_{\mathcal{L}} \text{Service}_B \Longrightarrow ((\text{Client} \bowtie_* \text{Probe}) \parallel \text{Client}[?s - 1]) \bowtie_{\mathcal{L}} \text{Service}_B$$

This gives rise to the full system equation:

$$(\text{Client}[m] \bowtie_{\mathcal{L}} \text{Service}_A) \parallel (((\text{Client} \bowtie_* \text{Probe}) \parallel \text{Client}[n - 1]) \bowtie_{\mathcal{L}} \text{Service}_B)$$

4.3 Average Response-time Example

We return to a simple model with a number of clients and services giving rise to the system equation:

$$\text{Client}[m] \bowtie_{\mathcal{L}} \text{Service}[n]$$

with $\mathcal{L} = \{\text{request}, \text{response}\}$. Note that we do not know much about the behaviour of the clients and the services only that they cooperate over the *request* and *response* activities. We wish to measure the response-time as observed by a single client. When calculating response-time quantiles we must separate out a particular client to observe and in the previous section we saw how we can do this using a probe and a transformation rule.

Suppose we instead wish to simply record the average response-time. This can be done with an application of Little's Law [16]. As before we can apply the response-time probe (*request:start*, *response:stop*) to a single client. However this increases the state-space size of the model, so if we are stretching our analysis as

far as possible (perhaps the reason we have dropped the requirement of obtaining a full response-time quantile profile) then we wish to avoid this. The average response-time can be calculated by attaching a probe to all clients. In the common case that the states along the passage for the client are distinguished anyway this will not increase the state space. We can still use Little’s Law; by dividing the number of local client probes within the passage by the throughput of the *request* activity we obtain the average response-time of the system. We can specify such a measurement as:

- Probe = *request:start, response:stop*
- Client[?*m*] \implies (Client \boxtimes Probe)[?*m*]

5 Case Study

To demonstrate these techniques we model an ecommerce site. Customers may browse the items for sale on a browse server which responds to search queries to display appropriate item pages. Once the user has browsed for a while they may decide to buy, in which case they are directed to the payment server. The payment server takes longer to provide a confirmation page because the method of payment must be confirmed. A PEPA model of this arrangement is shown in Figure 2. In this model all users eventually buy something which is arguably unrealistic. If we wished to avoid this we could have two classes of users.

The system can be configured by altering the following model parameters:

$\boxed{\text{users}}$ the average number of users within the system,

$\boxed{\text{b}}$ the number of browse servers,

$\boxed{\text{p}}$ the number of payment servers, this will generally be less than the number of browse servers since fewer payments than browse requests are made,

$\boxed{\text{r_browse}}$ the rate at which a user requests a new page,

$\boxed{\text{r_pay}}$ the rate at which a user decides to purchase an item and requests confirmation of payment, this will be somewhat less than the rate at which users browse pages since a user will typically browse several pages before deciding to buy,

$\boxed{\text{r_send_page}}$ the rate at which a browse server can send pages, and finally,

$\boxed{\text{r_send_confirm}}$ the rate at which a payment server can process payments this will generally be somewhat slower than r_send_page since the payment server must contact the financial institution for confirmation of available funds.

5.1 CTMC Results

With a small number of users it is possible to analyse this model using compilation from the PEPA model to a continuous-time-Markov-chain. We set the

$$\begin{array}{ll}
\text{User} & \stackrel{\text{def}}{=} (\text{browse}, r_{\text{browse}}).\text{Waiting} \\
& + (\text{pay}, r_{\text{pay}}).\text{Paying} \\
\text{Waiting} & \stackrel{\text{def}}{=} (\text{sendPage}, \top).\text{User} \\
\text{Paying} & \stackrel{\text{def}}{=} (\text{sendConfirm}, \top).\text{User} \\
\\
\text{BrowseServer} & \stackrel{\text{def}}{=} (\text{sendPage}, r_{\text{send_page}}).\text{BrowseServer} \\
\text{PaymentServer} & \stackrel{\text{def}}{=} (\text{sendConfirm}, r_{\text{send_confirm}}).\text{PaymentServer} \\
\\
\text{Servers} & \stackrel{\text{def}}{=} \text{BrowseServer}[b] \parallel \text{PaymentServer}[p] \\
\text{System} & \stackrel{\text{def}}{=} \text{User}[\text{users}] \underset{\mathcal{L}}{\bowtie} \text{Servers} \\
\text{where} & \\
\mathcal{L} & = \{\text{sendPage}, \text{sendConfirm}\}
\end{array}$$

Figure 2: A PEPA model of an online media distribution service.

numbers of users with the rate $\text{users} = 5$, to correspond to this we also keep the number of servers of each kind low with $b = p = 2$.

We performed three passage analyses over this model. The first two were the response-times as observed by a single user for the browse and payment servers. For the browse server we must measure between the activities `browse` and `sendPage` but only those performed by a single user. Similarly for the response-time of the payment server we measure between the activities `pay` and `sendConfirm`. The first probe is:

- Probe = `browse:start, sendPage:stop`
- $\text{User}[?n] \implies (\text{User} \underset{*}{\bowtie} \text{Probe}) \parallel \text{User}[?n - 1]$

The second probe is similarly defined and attached as:

- Probe = `pay:start, sendConfirm:stop`
- $\text{User}[?n] \implies (\text{User} \underset{*}{\bowtie} \text{Probe}) \parallel \text{User}[?n - 1]$

The results of these two analyses are shown in graph (a) of Figure 3 along with the results of the third passage analysis. This seeks to analyse the time taken from a user first browsing until they are sent a confirmation of their purchase. This uses the same probe placement transformation as the previous two analyses but measures between a `browse` activity and a `sendConfirm` activity using this `(browse:start, sendConfirm:stop)` probe specification. From the results we see that, as expected, the browsing response-time is very fast, the payment confirmation takes a little longer but is reasonable and has a high probability of completing within 10 time-units. Each user however has less than a forty-percent chance of completing the passage from first browse to payment confirmation within ten time units, since the user will on average browse ten times before initiating a payment which must then complete at the slower rate.

5.2 ODE Results

The CTMC analysis provides detailed results but suffers from the well-known state-space explosion problem. As we increase the number of users of the system

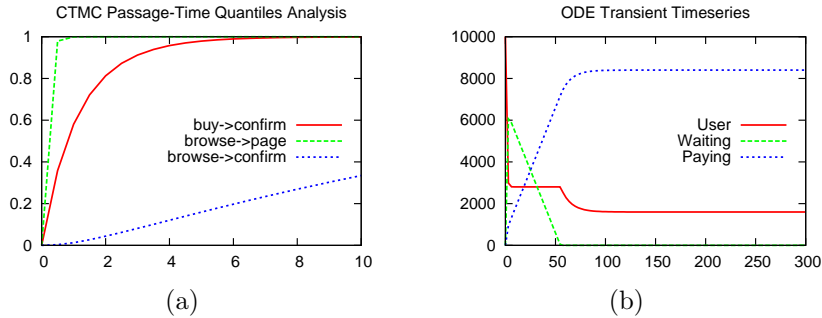


Figure 3: Graph (a) depicts the cumulative distribution function of three passages using CTMC analysis with a small number of clients. Graph (b) depicts the time-series analysis with a large number of clients using ODE analysis to plot the population of components against time.

the state space of the resulting CTMC grows exponentially. Very quickly we reach the point at which analysis becomes very expensive and then infeasible. An alternative method of analysis involves the translation of the PEPA model into a set of ordinary differential equations as proposed by Hillston [10]. Using this method we are able to analyse the same model with many more users as well as more servers. We initially set the number of users with $\text{users} = 500$. The number of server *processes* now becomes the number of physical machines multiplied by the number of threads running on each server machine. Therefore the number of server components are: $b = 8 \times \text{browse_threads}$ and $p = 8 \times \text{payment_threads}$, where browse_threads and payment_threads become additional rate parameters which may be ranged over. We speculated that a payment server would have fewer threads and so we set our initial values to; $\text{browse_threads} = 50$ and $\text{payment_threads} = 20$.

When the resulting set of ODEs are solved we obtain a *time-series* mapping which plots the population size of each kind of component in the model against time. For our initial values this gives rise to the graph (b) on the right of Figure 3. Initially all of the users are in the User state but very quickly the users perform one of the browse or pay activities to move into one of the two states Waiting or Paying. From this analysis alone we can see that the capacity of the browse servers is high enough and as a result, after an early spike the number of users waiting on the browse server to respond becomes close to zero. When this occurs the behaviour of the Paying and User populations change, but these too become stable and we see that the majority of users are waiting on a payment server to respond suggesting that we should increase the capacity of the payment servers.

In order to obtain steady-state results from this kind of analysis we continue to solve the ODEs for larger and larger times until the population sizes become stable as they have in our example in graph (b) of Figure 3 by around time 100. At this point we have reached the steady-state of the model, and we know that in the long run this will be the average population sizes of the components of the model.

We desire a measure of the responsiveness of the system, but as of yet there is no way to obtain the passage-time probability distributions as we have for

the CTMC analysis. However as we mentioned in Section 4.3 it is possible to use the steady-state to calculate the average response-time.

Here we focus on the response-time of the payment server however the same kind of analysis can be used to analyse the responsiveness of the browse server or the user time from first browse to completed payment. Our approach will be to attach a request/response style probe to every single user. We then take the steady-state population of all the local probes which are in the Running state. That is, those probes which have observed a pay without yet observing an associated sendConfirm. This in turn indicates how many users are waiting for a response from the payment server. With this value we take the steady-state throughput of the activity which begins a passage (i.e. the pay activity). By dividing the number of users waiting on a response by the throughput of the pay we calculate the average response-time of the payment servers as observed by a single user. The probe specification is therefore:

- Probe = pay:start, sendConfirm:stop
- $\text{User}[?n] \implies (\text{User} \bowtie_{*} \text{Probe})[?n - 1]$

In this example we could just take the population size of the user components in the ‘Paying’ state. However we use stochastic probes to guard against future modifications of the model which may add users states between the beginning and ending events of the passage in question. Indeed for analysing the time from first browse to eventual purchase there is no user state that corresponds to “within the passage” and hence we would require a probe component.

Unfortunately the attachment of a probe component to every user component translates the model into a format which is not suitable for translation into ODEs. All process arrays must be arrays of sequential components and cannot contain cooperations. The solution is to apply a partial evaluation of the $(\text{User} \bowtie_{*} \text{Probe})$ component to obtain a sequential component which is isomorphic to the original cooperation. This process has been described in [11] and may be automated.

We make the reasonable assumption that the response-time is dependent upon how many users are attempting to access the system. With this in mind we perform sensitivity analysis over the number of users, by varying the number of users while retaining all other rates constant and using the ODE analysis to calculate the response-time of both the ‘browse’ and ‘pay’ requests. The results are shown in the graph (a) on the top left of Figure 4. This plots the average response-time of both the browse and pay requests. We note that there is spare capacity on the browse servers because the response-time is very low and remains so even as the number of users is increased. However as the number of users increase this has significant impact on the response-time of users waiting on confirmation from a payment server. This suggests that the system may be improved by redeploying one or more browse servers as payment servers as we do in the following subsection.

5.3 Redeployment

We wish to predict how robust the system is to changes in its operating environment. In particular to increases in the number of users or the number of users who are making purchases. Since there are more than one of each kind of server

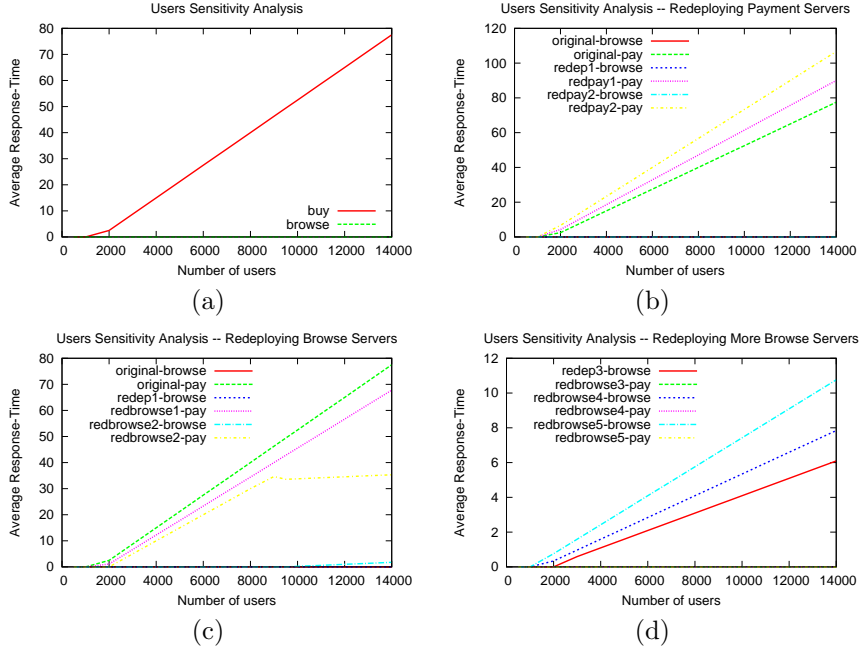


Figure 4: Sensitivity analysis for the number of users based on the number of redeployed servers.

the system can attempt to combat a changing environment by redeploying one kind of server as another. We may try these transformations:

$\text{BrowseServer}[?m] \parallel \text{PaymentServer}[?n]$ \implies $\text{BrowseServer}[?m - \text{browse_threads}] \parallel \text{PaymentServer}[?n + \text{payment_threads}]$
$\text{BrowseServer}[?m] \parallel \text{PaymentServer}[?n]$ \implies $\text{BrowseServer}[?m + \text{browse_threads}] \parallel \text{PaymentServer}[?n - \text{payment_threads}]$

The first transformation redeploys a browse server as a payment server and the second redeploys a payment server as a browse server. Of course more than one server may be redeployed by changing the expressions, for example two servers in: $[?m + (2 \times \text{browse_threads})]$.

The results in graphs (b), (c) and (d) of Figure 4 show the effect on the response-time of both the ‘browse’ and ‘pay’ requests for varying numbers of redeployments together with the initial results where there are no redeployments reshown for comparison. As a sanity check graph (b) redeploys payment servers as browse servers. Since there was already enough browse server capacity the browsing users gain no benefit from this as their response-time is very fast before the redeployments. However the response-time of the payment requests is worsened with more redeployment of payment servers.

Graph (c) shows the attempt to more evenly distribute the server capacity by redeploying one or two browse servers as payment servers. These results are encouraging as even at large numbers of users the browse response-time is

still not significantly affected but we have managed to reduce the response-time of the payment requests. This suggests that the system should by default be configured without an even number of servers but with ten payment servers and six browse servers.

Flushed with the success of redeploying two payment servers graph (d) now shows what happens when we redeploy further browse servers as payment servers. Here though we have started to go too far, at this level of redeployment the payment server is always fast enough but the browse server response-time starts to increase. From this we tentatively conclude that the correct configuration should be two or three browse servers redeployed as payment servers but of course further analysis of throughput of actual purchases would be required in practice.

Finally from all of the results we note that regardless of the configuration the system copes well whenever the number of users is less than 2000 which might suggest that some servers could be turned off at times of low-demand.

6 Conclusions

We have developed a language for describing transformations over PEPA models. This has proven to have two uses, the first of which is as a way to programmatically derive families of models from one original model and thereby systematically analyse the effect of changing the model. This allows the modeller to analyse the robustness of a system to changes in its operating environment including the modification of the system itself. The second use is as a probe placement language. The use of local probe components which can restrict their observations to a sub-component of the entire model is crucial for accurate and detailed performance measurements; we have found that this is particularly the case for passage-time analysis. Until now formal placement of the probe component automatically derived from the probe-specification has been ad-hoc and limited. The transformation language described in this paper provides an essential component of the probe-based query specification framework. That we have been able to fill this gap using an otherwise useful transformation language reduces the amount of learning necessary for a new user to obtain the analyses they desire.

We have shown that by utilising this probe placement mechanism the same probe specification may be used to analyse two different models for both complete response-time distributions and average response-time changing only the placement transformation rule. Using this we analysed response-time quantiles in a model which could be compiled to a CTMC and then use the same probe for analysing the average response-time of a model with a much larger component population through compilation to ordinary differential equations.

We also provide further usage of two separate compilation techniques for PEPA models and hence further evidence of the utility of this approach seen before in [11].

The work described here has been fully implemented in the International PEPA Compiler [17, 18] which is available for free download from the PEPA Web site at <http://www.dcs.ed.ac.uk/pepa>.

In the future we hope to add some computed transformations to our language of transformations. These include such transformations as partial evaluation and

concatenation of non-cooperating sequential activities. As we have mentioned above partial evaluation can convert a model inappropriate for fluid-flow analysis into one which may be analysed. This could be controlled by the user through the use of transformation functions, such as:

$$P \underset{?a}{\boxtimes} Q \implies PE(P \underset{?a}{\boxtimes} Q)$$

where the function $PE(P)$ performs partial evaluation over the process P . Some models may be simplified into models which then admit more efficient solution techniques such as transforming a model into a queuing model as done in [19]. Although partial evaluation always transforms the model into an equivalent model other simplifying transformations may change the behaviour of the model and hence must only be applied under direct human instruction. A good example is the concatenation of sequential activities in order to make the model less *stiff*. A stiff model has some very large rates which hinders solution but if we are interested in only average behaviour then we can transform a sequential process such as: $(a, r_1).(b, r_2).(c, r_3).P$ into $(abc, 1/(\frac{1}{r_1} + \frac{1}{r_2} + \frac{1}{r_3})).P$ provided that the process is not involved in any cooperation over any of the three activities concerned. The combined rate will be low if any of the three original rates are low and hence we may obtain a solution in much less time. Once again we would use a transformation rule involving a function such as:

$$P \implies Coalesce(P, [a, b, c])$$

where the function $Coalesce(P, actions)$ coalesces any sequence in the process P involving only the given actions.

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)). The ipc/Hydra tool chain has been developed in co-operation with Jeremy Bradley, Will Knottenbelt and Nick Dingle of Imperial College, London.

References

- [1] Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
- [2] 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, ACM Press (2004) 49–58
- [3] Clark, A., Gilmore, S.: State-aware performance analysis with eXtended Stochastic Probes. In Thomas, N., Juiz, C., eds.: Proceedings of the 5th European Performance Engineering Workshop (EPEW 2008). Volume 5261 of LNCS., Palma de Mallorca, Spain, Springer (2008) 125–140
- [4] Hillston, J.: The nature of synchronisation. In Herzog, U., Rettelbach, M., eds.: Proceedings of the Second International Workshop on Process Algebras and Performance Modelling, Erlangen (1994) 51–70
- [5] Hillston, J.: Tuning systems: From composition to performance. The Computer Journal **48**(4) (2005) 385–400 The Needham Lecture paper.

- [6] Hillston, J.: Process algebras for quantitative analysis. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05), Chicago, IEEE Computer Society Press (2005) 239–248
- [7] Tomasik, J., Hillston, J.: Transforming PEPA models to obtain product form bounds. Technical Report EDI-INF-RR-0009, Laboratory for Foundations of Computer Science, The University of Edinburgh (2000)
- [8] Gilmore, S., Grant-Duff, Z., Harrison, P., Hillston, J.: Systematic transformations to find quasi-reversible structures in PEPA models. In: Proceedings of the Workshop on Process Algebra and Stochastically Timed Activities (PASTA 2002), Edinburgh (2002) 1–16
- [9] Harrison, P., Hillston, J.: Exploiting quasi-reversible structures in Markovian process algebra models. In Gilmore, S., Hillston, J., eds.: Proceedings of the Third International Workshop on Process Algebras and Performance Modelling, Special Issue of *The Computer Journal*, 38(7) (1995) 510–520
- [10] Hillston, J.: Fluid flow approximation of PEPA models. In: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, Torino, Italy, IEEE Computer Society Press (2005) 33–43
- [11] Clark, A., Duguid, A., Gilmore, S., Tribastone, M.: Partial evaluation of PEPA models for fluid-flow analysis. In Thomas, N., Juiz, C., eds.: Proceedings of the 5th European Performance Engineering Workshop (EPEW 2008). Volume 5261 of LNCS., Palma de Mallorca, Spain, Springer (2008) 2–16
- [12] Hayden, R., Bradley, J.T.: Fluid semantics for passive stochastic process algebra cooperation. In: VALUETOOLS'08, 3rd International Conference on Performance Evaluation Methodologies and Tools. (2008)
- [13] Clark, A., Gilmore, S., Tribastone, M.: Service-level agreements for service-oriented computing. In: Proceedings of the 19th International Workshop on Algebraic Development Techniques (WADT 2008), Pisa, Italy (2008) To appear.
- [14] Clark, A., Gilmore, S., Tribastone, M.: Scalable analysis of scalable systems. In: Proceedings of Fundamental Approaches to Software Engineering (FASE 2009), York, England (2009) To appear.
- [15] 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.: Proceedings of the Third International Conference on Trustworthy Global Computing (TGC'07). Volume 4912 of LNCS., Springer-Verlag (2008) 222–239
- [16] Little, J.D.C.: A proof of the queueing formula $l = \lambda w$. *Operations Research* **9** (1961) 380–387
- [17] 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, IEEE Computer Society Press (2003) 344–351
- [18] 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), IEEE (2007) 55–56
- [19] Zhao, Y., Thomas, N.: Approximate solution of a pepa model of a key distribution centre. In: SIPEW '08: Proceedings of the SPEC international workshop on Performance Evaluation, Berlin, Heidelberg, Springer-Verlag (2008) 44–57