Visualisation for Stochastic Process Algebras: The Graphic Truth

Michael J. A. Smith[1] and Stephen Gilmore[2]

[1] Department of Informatics and Mathematical Modelling
Danmarks Tekniske Universitet, Lyngby, Denmark
mjas@imm.dtu.dk
[2] Laboratory for Foundations of Computer Science
University of Edinburgh, Edinburgh, United Kingdom
Stephen.Gilmore@ed.ac.uk

Abstract. There have historically been two approaches to performance modelling. On the one hand, textual language-based formalisms such as stochastic process algebras provide portable and scalable solutions to compositional modelling. On the other hand, graphical formalisms such as stochastic Petri nets and stochastic activity networks provide an automaton-based view of the model, which may be easier to visualise, at the expense of portability. In this paper, we argue that we can achieve the benefits of both approaches by *generating* a graphical view of a stochastic process algebra model, which is synchronised with the textual representation, giving the user has two ways in which they can interact with the model.

We present a tool, as part of the PEPA Eclipse Plug-in, that allows the components of models in the Performance Evaluation Process Algebra (PEPA) to be visualised in a graphical way. This also provides a natural interface for labelling states in the model, which integrates with our interface for specifying and model checking properties in the Continuous Stochastic Logic (CSL). We describe recent improvements to the tool in terms of usability and exploiting the visualisation framework, and discuss some of the general features of the implementation that could be used by other tools. We illustrate the tool using an example based on a model of a financial web-service application.

1 Introduction

It is often said that seeing is believing. Even though we know from biology that the eye can be tricked in all manner of ways, most people will agree that being able to see — or *visualise* — something with their own eyes adds great weight to their belief in it. This is true in performance modelling, just as much as in the real world. Unlike a computer program, which implements a specification and therefore can be tested for correctness, a performance model is often a specification in and of itself. This leads to a big problem — how do we *convince* ourselves that the model we have written is really the same as the model we intended to write?

There are many approaches to performance modelling, but the use of *language-based* formalisms such as *stochastic process algebras* [14, 16] have been particularly successful. In addition to being natural for computer scientists, who are used to programming in linear, text-based languages, they have the advantage of portability — we do not require a special tool to view or edit the model. A disadvantage, however, is that it can be difficult to visualise the behaviour of the model — for example, a small typo in the model can lead to strange and unintended behaviour, but can easily go unnoticed.

An alternative approach is to use *graphical* formalisms for performance modelling, such as stochastic Petri nets [4] and stochastic activity networks [19]. Since these are automata-based formalisms, it is easy to visualise the structure and behaviour of components in the model. Whilst several highly successful tools make use of such formalisms — for example PIPE [5] and Möbius [11] — they suffer from some limitations. Most notably, *portability* of the model between tools, and *scalability*, since the larger a component is, the more difficult it is to manage and 'see as a whole'.

The contribution of this paper is to bring these two approaches together, in a tool that supports *two different views* of the same model. We present an extension to the PEPA Eclipse Plug-in [28] that allows performance models in the Performance Evaluation Process Algebra (PEPA) [14] to be presented *graphically*. This is useful not only for visualising the model, but also as an intuitive interface for *abstracting* it, and for specifying *performance properties* we would like to verify.

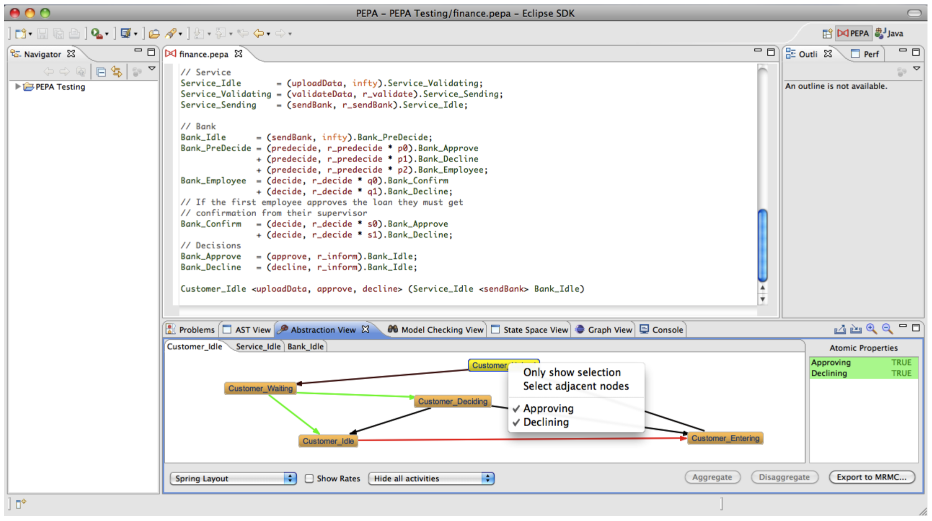
Since we have already presented a summary of our tool in [22], it is important to clarify the purpose of this paper. Our focus here is not on the back-end of the tool — namely, on the support for compositional abstractions and model checking [23, 24] — but on the *novel user interfaces* that we have developed. We present some significant improvements in features and usability compared with [22], and moreover describe the implementation details of our front-end, to allow the principles to be applied to other tools based on stochastic process algebra. Figure 1 shows a screenshot of the plug-in.

We begin in Section 2 by introducing the PEPA language, along with a running example based on a financial web service case study. We then motivate the need for visualisation of PEPA models in Section 3, before introducing the visualisation features of the PEPA Eclipse Plug-in. In Section 4, we describe how PEPA models can be visualised in a graphical way, along with how this interface can be used for specifying abstractions of the model and labelling states. In Section 5, we then present the interface for constructing performance properties (in the Continuous Stochastic Logic (CSL) [3]), which ensures that the user can only enter syntactically valid properties. We discuss the implementation details, such as the data structures that allow us to share information between different interfaces, in Section 6, before discussing related work in Section 7 and concluding in Section 8.

2 Modelling in PEPA

The Performance Evaluation Process Algebra (PEPA) [14] is a widely used language for performance modelling and analysis, which allows models to be built compositionally. PEPA models are built out of *components*, which run in parallel and can perform *activities*. An activity (a, r) is a pair consisting of an *action type* $a \in \mathcal{A}$, and a *rate* $r \in \mathbb{R}_{\geq 0} \cup \{\top\}$. The rate parameterises an exponential distribution that describes the duration of the activity. The special rate $\top$ denotes a *passive rate*, meaning that another component must determine the rate of the activity. The syntax of PEPA is as follows:

$$\begin{aligned} C_S &:= (a, r).C_S \mid C_S + C_S \mid A \\ C_M &:= C_S \mid C_M \underset{L}{\bowtie} C_M \mid C_M / L \end{aligned}$$



```
// Service
Service_Idle       = (uploadData, infty).Service_Validating;
Service_Validating = (validateData, r_validate).Service_Sending;
Service_Sending    = (sendBank, r_sendBank).Service_Idle;

// Bank
Bank_Idle      = (sendBank, infty).Bank_PreDecide;
Bank_PreDecide = (predecide, r_predecide * p0).Bank_Approve
               + (predecide, r_predecide * p1).Bank_Decline
               + (predecide, r_predecide * p2).Bank_Employee;
Bank_Employee  = (decide, r_decide * q0).Bank_Confirm
               + (decide, r_decide * q1).Bank_Decline;
// If the first employee approves the loan they must get
// confirmation from their supervisor
Bank_Confirm   = (decide, r_decide * s0).Bank_Approve
               + (decide, r_decide * s1).Bank_Decline;
// Decisions
Bank_Approve   = (approve, r_inform).Bank_Idle;
Bank_Decline   = (decline, r_inform).Bank_Idle;

Customer_Idle <uploadData, approve, decline> (Service_Idle <sendBank> Bank_Idle)
```

Fig. 1. The PEPA Eclipse Plug-in, showing the editor and the abstraction view

Here, we call C_S *sequential components*, and C_M *model components*. A PEPA model is constructed by defining a collection of sequential components, along with a model component called the *system equation*, which describes the initial configuration of the model. The PEPA combinators are as follows:

Prefix	$(a, r).C$	The component can carry out an activity (a, r) to become C.
Choice	$C_1 + C_2$	The component may behave as either C_1 or C_2, according to the first that completes an activity (a race condition).
Cooperation	$C_1 \underset{L}{\bowtie} C_2$	C_1 and C_2 synchronise over the actions in L (the cooperation set). For activities whose type is not in L, the two components proceed independently. Otherwise, they must perform the activity together, at the rate of the slowest component.
Hiding	C/L	The component behaves as C, except that activities with an action type in L are hidden, and cannot be synchronised over.
Constant	$A \overset{def}{=} C$	Component C has the name A.

PEPA has an operational semantics, which maps a model onto a labelled multi-transition system, from which a continuous-time Markov chain (CTMC) is derived [14]. If we want to operate on the underlying CTMC of a model in a *compositional* way, however, it is more useful to use an alternative semantics based on a *Kronecker representation*. This was first introduced in [15], and developed further in [23,24]. It was proven in [23] that the reachable state space of the CTMC given by the Kronecker semantics is isomorphic to that given by the original semantics of PEPA in [14].

To specify the CTMC of a PEPA model in a compositional way, we first need to define the notion of a *CTMC component*:

Definition 1. *A* CTMC component *is a tuple* $(S, r, \boldsymbol{P}, L)$*, where* S *is a finite non-empty set of states,* $r : S \to \mathbb{R}_{\geq 0} \cup \{ \top \}$ *assigns a rate (or* $\top$*) to each state,* $\boldsymbol{P} : S \times S \to [0, 1]$ *assigns a probability distribution over* S *to each state* $s \in S$*, and* $L : S \to AP$ *is a labelling function (*AP *is a finite set of atomic propositions). We require for all* $s \in S$ *that* $\sum_{s' \in S} \boldsymbol{P}(s, s') = 1$.

Note that if all the rates are active, a CTMC component is just a standard CTMC.

We construct a PEPA model by *composing* CTMC components. To do this, we use two composition operators, $\circledast$ and $\odot$, which correspond to synchronised and independent parallel composition respectively. For CTMC components $M_1 = (S_1, r_1, \boldsymbol{P}_1, L_1)$ and $M_2 = (S_2, r_2, \boldsymbol{P}_2, L_2)$, these are defined as:

$$M_1 \circledast M_2 = (S_1 \times S_2, \min\{ r_1, r_2 \}, \boldsymbol{P}_1 \otimes \boldsymbol{P}_2, L_1 \times L_2)$$

$$M_1 \odot M_2 = (S_1, r_1, \boldsymbol{P}_1, L_1) \circledast (S_2, r'_\top, \boldsymbol{I}, L_2) + (S_1, r'_\top, \boldsymbol{I}, L_1) \circledast (S_2, r_2, \boldsymbol{P}_2, L_2)$$

where we define $\min\{ r_1, r_2 \}(s_1, s_2) = \min\{ r_1(s_1), r_2(s_2) \}$, $(L_1 \times L_2)(s_1, s_2) = L_1(s_1) \cap L_2(s_2)$. $r_\top(s) = \top$ for all s, and $\boldsymbol{I}$ is the identity matrix ($\boldsymbol{I}(s_1, s_2) = 1$ if $s_1 = s_2$ and 0 otherwise). $\otimes$ is the standard Kronecker product of two matrices [20].

For two CTMC components $M_1 = (S, r_1, \boldsymbol{P}_1, L)$ and $M_2 = (S, r_2, \boldsymbol{P}_2, L)$ with the same state space S and labelling function L, the addition used in the previous equation is defined as follows:

$$M_1 + M_2 = \left(S, r_1 + r_2, \frac{r_1}{r_1 + r_2} \boldsymbol{P}_1 + \frac{r_2}{r_1 + r_2} \boldsymbol{P}_2, L \right)$$

where we define $(r_1 + r_2)(s) = r_1(s) + r_2(s)$, $\frac{r_i}{r_1+r_2}(s) = \frac{r_i(s)}{r_1(s)+r_2(s)}$, $i \in \{ 1, 2 \}$, and $(r\boldsymbol{P})(s_1, s_2) = r(s_1)\boldsymbol{P}(s_1, s_2)$.

The Kronecker semantics of PEPA is as follows. For a PEPA sequential component C, we use the operational semantics in [14] to derive a CTMC component: $[\![C]\!]^{PEPA} = (S, r, \boldsymbol{P}, L)$. Technically, the labelling function L is not given as part of the model, but we will show how to define it using the PEPA Eclipse Plug-in, in Section 4. We can similarly define $[\![C]\!]^{PEPA}_a = (S, r_a, \boldsymbol{P}_a, L)$ to be the CTMC component over the same state space S, where only the contribution of activities of action type a is considered (if a state s cannot perform an activity of type a, $r_a(s) = 0$).

Definition 2. *The CTMC induced by a PEPA model* C *is:*

$$[\![C]\!] = \sum_{a \in Act(C)} [\![C]\!]_a$$

where $Act(C)$ *is the set of all action types that occur in* C *(both synchronised and independent), and* $[\![C]\!]_a$ *is as follows:*

$$\begin{array}{ll} [\![C]\!]_a & = [\![C]\!]^{PEPA}_a \quad \text{if } C \textit{ is a sequential component} \\ [\![C_1 \bowtie_{\mathcal{L}} C_2]\!]_a & = \begin{cases} [\![C_1]\!]_a \circledast [\![C_2]\!]_a & \text{if } a \in \mathcal{L} \\ [\![C_1]\!]_a \odot [\![C_2]\!]_a & \text{if } a \notin \mathcal{L} \end{cases} \end{array}$$

$$\begin{array}{ll}
\mathit{Customer\_Idle} & = (\mathit{request}, r_{\mathit{request}}).\mathit{Customer\_Entering} \\
\mathit{Customer\_Entering} & = (\mathit{enterData}, r_{\mathit{enter\_data}}).\mathit{Customer\_Upload} \\
\mathit{Customer\_Upload} & = (\mathit{uploadData}, r_{\mathit{upload}}).\mathit{Customer\_Waiting} \\
\mathit{Customer\_Waiting} & = (\mathit{approve}, r_{\mathit{inform}}).\mathit{Customer\_Idle} \\
 & + (\mathit{decline}, r_{\mathit{inform}}).\mathit{Customer\_Deciding} \\
\mathit{Customer\_Deciding} & = (\mathit{reapply}, r_{\mathit{reapply}} \times t_0).\mathit{Customer\_Entering} \\
 & + (\mathit{reapply}, r_{\mathit{reapply}} \times t_1).\mathit{Customer\_Idle} \\
 & \\
\mathit{Service\_Idle} & = (\mathit{uploadData}, r_{\mathit{upload}}).\mathit{Service\_Validating} \\
\mathit{Service\_Validating} & = (\mathit{validateData}, r_{\mathit{validate}}).\mathit{Service\_Sending} \\
\mathit{Service\_Sending} & = (\mathit{sendBank}, r_{\mathit{sendBank}}).\mathit{Service\_Idle} \\
 & \\
\mathit{Bank\_Idle} & = (\mathit{sendBank}, r_{\mathit{sendBank}}).\mathit{Bank\_PreDecide} \\
\mathit{Bank\_PreDecide} & = (\mathit{predecide}, r_{\mathit{predecide}} \times p_0).\mathit{Bank\_Approve} \\
 & + (\mathit{predecide}, r_{\mathit{predecide}} \times p_1).\mathit{Bank\_Decline} \\
 & + (\mathit{predecide}, r_{\mathit{predecide}} \times p_2).\mathit{Bank\_Employee} \\
\mathit{Bank\_Employee} & = (\mathit{decide}, r_{\mathit{decide}} \times q_0).\mathit{Bank\_Confirm} \\
 & + (\mathit{decide}, r_{\mathit{decide}} \times q_1).\mathit{Bank\_Decline} \\
\mathit{Bank\_Confirm} & = (\mathit{decide}, r_{\mathit{decide}} \times s_0).\mathit{Bank\_Approve} \\
 & + (\mathit{decide}, r_{\mathit{decide}} \times s_1).\mathit{Bank\_Decline} \\
\mathit{Bank\_Approve} & = (\mathit{approve}, r_{\mathit{inform}}).\mathit{Bank\_Idle} \\
\mathit{Bank\_Decline} & = (\mathit{decline}, r_{\mathit{inform}}).\mathit{Bank\_Idle}
\end{array}$$

$$\mathit{Customer\_Idle} \underset{\{\,\mathit{uploadData}, \mathit{approve}, \mathit{decline}\,\}}{\bowtie} \left(\mathit{Service\_Idle} \underset{\{\,\mathit{sendBank}\,\}}{\bowtie} \mathit{Bank\_Idle}\right)$$

Fig. 2. A PEPA model of a financial web service application

Example: As a running example for the remainder of this paper, consider the PEPA model in Figure 2. This is a model of a financial services case study from the SENSORIA project — a five-year EU-funded project on software engineering for service-oriented computing [21]. The project brought together a large number of European universities and research centres together with four industrial partners, one of whom was a European bank engaged in business-to-business operation. The bank explained the process by which loans are awarded to businesses using a procedure that must provide a reliable workflow to guard against fraud and meet constraints imposed on fiscal and monetary transactions by law.

The PEPA model presented here describes this workflow in terms of a customer using a service portal and interacting with the bank where employees approve or decline loans subject to managerial approval. Structurally, the model is a typical idiomatic PEPA model, with a small number of sequential components, which may be replicated to make larger instances of the problem. These sequential components are brought together in a parallel composition which requires them to co-operate on shared activities (such as $\mathit{uploadData}$) and to proceed independently on other activities (such as the decide activity approving the loan request).

Some components have a relatively complex workflow with multi-way branching and loops taking them to different entry points in the workflow. Each component is cyclic so that the model has a meaningful steady state solution, and describes an unending process with infinite behaviour. The visualisation capabilities of the PEPA Eclipse plug-in were very helpful in enabling us to communicate the meaning of the model to partners in the project who were not familiar with process calculi and stochastic processes.

A more complete description of the SENSORIA Finance Case Study appears in [9].

3 The Argument for Visualisation

The approach adopted here for visualising PEPA models differs from the approach taken in graphical modelling formalisms such as Petri nets and Stochastic Activity Networks where the visual representation and layout of the model is central. In those formalisms the modeller most often creates a manual layout of the model. (This is the case for the PIPE Petri net editor [5] and the SAN editor of Möbius [11].) Other, mostly textual, representations of the model such as XML or program source code are generated from the graphical representation source.

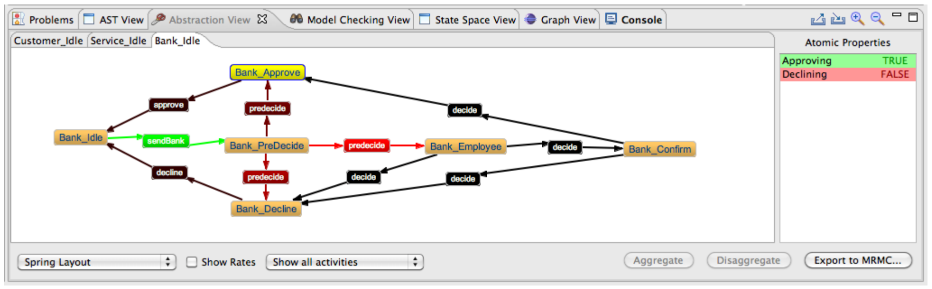
Here instead, our wish was that the textual representation of the PEPA model is considered to be the model source, which has primary importance, and that graphical representations are automatically derived from this and have secondary importance. We have used automatic layout algorithms to generate a first attempt at a layout which can then be manually improved by the user according to their aesthetic sensibilities and tastes. The manually improved version is automatically saved in order that this work does not need to be redone after every edit. We believe that this is a good pragmatic compromise between a fully manual and a fully automatic approach to visualisation.

It was important to us than the graphical representation of the PEPA model should be based on the components which PEPA uses to structure large models. The visualisation of the model is only helpful if the user can meaningfully interpret the visualisation. From earlier work [8], we have seen that each component of the model is a coherent unit of behaviour and we believe that selecting this as the way to structure the visualisation is always likely to give a more comprehensible visualisation than any which is based on a lower level of abstraction such as the underlying CTMC.

Given this perspective on the problem, we are able to reflect in the visualisation of the components the knowledge obtained from the model about activities which are passive, or fast, or slow. Given that we are representing a stochastic process algebra model, we feel that it is essential that rates should play some part in the visual representation. Without this, the visualisation is only helping to detect problems in model behaviour. It is important to remember that all analysis results derived from the model depend upon the rate parameters used and that an error in a rate parameter definition can be easily made with a simple typo which perhaps just puts the decimal point in the wrong place. Similarly, omitting parameter information from an activity — by declaring it erroneously as passive rather than active — can lead to different performance results being computed. Both of these errors can be very easily seen with our current visualisation because, for each component, fast activities are coloured differently from slow ones, and passive activities are coloured differently again.

4 Visualisation of PEPA Models

The PEPA Eclipse Plug-in allows us to edit and analyse PEPA models using the popular Eclipse framework [1]. This separates the user interface into two main parts. The *edi-*



Atomic Properties	
Approving	TRUE
Declining	FALSE

Spring Layout

Show all activities

Fig. 3. The abstraction view

tor window is where the PEPA model is displayed, and can be edited. This is basically a text editor, with additional features such as syntax highlighting and identification of parse errors. Alongside the editor are a number of *views*, which are used to display information about the model, and as an interface for generating information (e.g. through analysis). This can be seen in Figure 1, where the editor is positioned centrally, with the views below it and to the right. In this section, we will look at the **abstraction view** (shown at the bottom of the screen), which provides a graphical interface for viewing PEPA models.

The idea of visualising PEPA models is not a new one, and was first proposed in [27]. Since PEPA is a compositional language, we can view each component in the model independently, as an automaton whose transitions are labelled by activities. In [27] it was suggested that we can do this by displaying the derivation graph of a component. We use a slightly different approach based on the Kronecker representation described in the previous section — this is entirely equivalent in terms of what the user sees, but the underlying data structure is more versatile, as we shall discuss in Section 6.

Figure 3 shows the abstraction view, displaying the *Bank* component from Figure 2. There are four essential features of this view:

1. **Visualising**: In the main panel of the view is a series of tabs, which display an automaton for each sequential component in the PEPA model. This allows a fast visualisation of the component as described, so that certain errors in the model can be seen immediately — for example, if the component is supposed to cycle between a number of states, we expect to see a cyclic automaton.

   Since an individual PEPA component typically has a small number of states, its automaton will usually be small enough to display clearly. However, for larger or more complicated components, we have a feature to display only certain states. Either we can manually select the states we are interested in, right-click, and select 'only show selection', or we can right-click on a blank area and select 'choose states to select...', which offers a dialog box where we can select the states by name.

   Active and passive transitions have different colours (red and green respectively) so that they can be more easily distinguished. Moreover, we display active transitions in varying shades of red, ranging from bright red (for the fastest transitions) to black (for the slowest transitions). From a drop-down box, we can select whether to label the transitions with their activities — either for all transitions or for only certain transitions (such as the passive ones, or the fastest active ones).

2. **Labelling**: On the right-side of the view is a list of atomic properties that are defined for the model, which can be referred to in performance properties. These can be thought of as *labels*, which identify a set of states in the model using a more human-readable name.

   To define a new label, we first select the states that we want to label, and then right-click in the atomic properties list, and select '*new property*'. We can change which properties a state is labelled with by right-clicking on it, which allows us to select, or deselect atomic properties. Clicking on a state will display (in the atomic properties list) which properties are true or false, and clicking on a property selects all the states that are labelled with it. Note that atomic properties must be specified compositionally.

3. **Abstracting**: In order to analyse the model, we may want to reduce its size by first *aggregating* certain states. In the back-end of the tool, compositional abstraction techniques based on abstract Markov chains [23] and stochastic bounds [24] are used, but the front-end interface is very simple. The user simply has to select the states they want to aggregate, and click the "aggregate" button. These states can subsequently only be selected as a unit, and moved (or labelled) together.

4. **Exporting**: We allow the model to be exported to the input format for the MRMC model checker [17]. If the model has not been abstracted, the output is a CTMC (consisting of a `.lab` and `.tra` file), otherwise the output is a CTMDP (a `.lab` and a `.ctmdpi` file). This means that MRMC can be used as an alternative to the in-built model checker in the plug-in. Note, however, that MRMC currently only supports time-bounded reachability properties for CTMDPs, whereas the in-built model checker supports all the CSL operators[3].

When we create a new PEPA model, or load a new model that we have not worked with before, the abstraction view uses an automated layout algorithm from the Zest library. The idea is to provide a rough initial layout that can be changed by the user to one of their liking. Activity labels are automatically placed so that multiple transitions between two states do not overlap with one another. The layout information and defined properties are *automatically saved* (to an XML format) by the tool, so that if we return to a model in the future, the abstraction view looks precisely as we left it.

An important feature of the abstraction view is that it is *robust* with regard to minor changes to the model. For example, if we add a new component in the system equation, the layout information for the existing components is preserved. If we add a new state to a sequential component, the node appears in the default location (the top-left corner of the view) and must be placed manually (or a layout algorithm re-applied), but the other nodes remain in their correct position. If the tool detects a new state for which it

[3] Except the time-bounded next operator, since this is not preserved after uniformisation.

has no information, it displays a warning, and sets all atomic propositions to be true for that state, by default.

5 Constructing Performance Properties

There are a number of ways to describe and analyse performance properties of a PEPA model. In many cases, we may want to directly analyse a simple property, such as the steady state probability of a set of states, or the throughput of a given action. In this case, the plug-in provides a simple interface for obtaining such information. Often, however, we want to ask more sophisticated questions, and so we need a more powerful language to describe it. The PEPA Eclipse Plug-in supports two ways of doing this: stochastic probes [10], which uses a regular-expression syntax if we are interested in the passage time distribution between two events, and Continuous Stochastic Logic (CSL) [3], which we describe here.

There are two types of CSL properties: state properties $\varPhi$, which concern a state in the model, and path properties φ, which concern a sequence of states (a path) in the model. Together, these allow the logic to specify many useful performance properties, which can be analysed in a systematic way using a model checker. The plug-in has a built-in CSL model checker but, as described in the previous section, it is also possible to export models to MRMC.

The syntax of CSL supported by the PEPA Eclipse Plug-in is as follows (where we include derived operators):

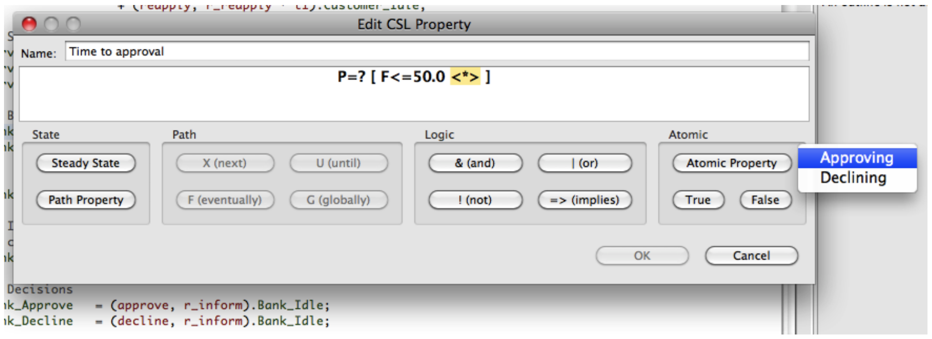
$$\begin{aligned}\varPhi &::= \texttt{tt} \mid \texttt{ff} \mid a \mid \varPhi \wedge \varPhi \mid \varPhi \vee \varPhi \mid \varPhi \Rightarrow \varPhi \mid \neg\varPhi \mid \mathcal{S}_{\trianglelefteq p}(\varPhi) \mid \mathcal{P}_{\trianglelefteq p}(\varphi)\\ \varphi &::= X\,\varPhi \mid \varPhi\, U^{I}\, \varPhi \mid F^{I}\, \varPhi \mid G^{I}\, \varPhi\end{aligned}$$

where $\trianglelefteq \in \{\leq, \geq\}$, $a \in AP$, $p \in [0,1]$, and $I = [a,b]$ is a non-empty interval over the reals, such that $a, b \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, and $a \leq b$.

Rather than give the formal semantics of CSL in this paper, we will consider the five types of state property that we most commonly construct:

Steady State	$\mathcal{S}_{\geq p}(\varPhi)$	In the long-run behaviour of the model, does $\varPhi$ hold with probability at least p?
Next	$\mathcal{P}_{\geq p}(X\,\varPhi)$	In the next state, does $\varPhi$ hold with probability at least p?
Until	$\mathcal{P}_{\geq p}(\varPhi_1\, U^{\leq t}\, \varPhi_2)$	Will $\varPhi_2$ become true no later than time t, and $\varPhi_1$ hold at all times until this point, with probability at least p?
Eventually	$\mathcal{P}_{\geq p}(F^{\leq t}\, \varPhi)$	Will $\varPhi$ become true no later than time t with probability at least p?
Globally	$\mathcal{P}_{\geq p}(G^{\leq t}\, \varPhi)$	Will $\varPhi$ always be true until time t with probability at least p?

At the top level, we also support *quantitative* CSL properties, of the form $\mathcal{S}_{=?}(\varPhi)$ and $\mathcal{P}_{=?}(\varphi)$, which return the actual probability of the given steady state or path property, rather than comparing it to a fixed value. These are very useful in practice.



Name: Time to approval

P=? [F<=50.0 <*>]

```
Decisions
k_Approve    = (approve, r_inform).Bank_Idle;
k_Decline    = (decline, r_inform).Bank_Idle;
```

Fig. 4. The CSL property editor

In general, since we support model checking of abstracted models (i.e. CTMDPs in addition to CTMCs), we use a three-valued variant of CSL. That is to say, a property can be true, false, or *maybe*. Similarly, a quantitative property returns an *interval* of probabilities — so, if we get $[0.1, 0.3]$, we know that the property holds of the unabstracted model with probability between 0.1 and 0.3. If a qualitative property returns 'maybe', or the interval of probabilities for a quantitative property is too wide, we should experiment with different abstractions to achieve better results. The theory underlying the tool [23, 24] guarantees the accuracy of the model checker, but the precision depends on the choice of abstraction.

To specify CSL properties and perform model checking, the plug-in provides a **model checking view**. This is basically just a table of CSL properties, which can be verified using the internal model checker. We allow saving and loading of properties, in an XML format, so that the same properties can be shared between different models. The most interesting feature, however, is our novel interface for constructing CSL properties.

There are a number of different ideas when it comes to helping a user specify a performance property. One idea, such as in performance trees [26], is to move away from a textual description, and to visualise the property in a more graphical, and hopefully more user-friendly way. The other idea is to provide a simple dialog box where the user types in the property in a logic like CSL, such as in PRISM [16]. Our approach lies somewhere between the two, in that we *do* expect the user to be expert enough to understand CSL, but we *do not* expect them to know the specific syntax used by the tool. To this end, our interface provides a lot of support to the user, preventing them from making syntax errors.

Figure 4 shows the CSL editor. We cannot type the property by hand, but instead are guided by the enabled buttons, corresponding to CSL terms that can be used in the current position. The buttons all have keyboard shortcuts so that experienced users can create properties more quickly. When we click on part of the property, the editor determines which term we clicked on, and highlights the entire term. We then have the option to substitute it with another term — if no term has yet been entered, or we delete a term, the placeholder '<*>' is seen. Numerical parts of the property can be edited directly, but they will only be accepted if syntactically correct.

The most useful feature of the editor is that it is linked to the abstraction view. When we click on the 'Atomic Property' button, we see a list of all the labels that have been defined for the model. This avoids us having to switch back and forth between the two views. Because the underlying data structures are shared between the two views (we will describe this in more detail in the next section), if we change the name of an atomic property in the abstraction view, it is immediately updated in the model checking view. The plug-in prevents us from deleting an atomic property if it is in use.

Since we can load CSL properties, the plug-in also has a built-in parser for CSL. Only syntactically correct properties will be loaded, and if any atomic properties are used that were not defined, a warning is given and they are replaced with 'true'.

6 Implementation Details

The key idea behind the implementation of the abstraction and model checking functionality of the plug-in lies in the Kronecker representation described in Section 2. Rather than explicitly constructing the state space of the PEPA model, we *only* construct the CTMC component for each sequential component and action type. These are only composed when we actually perform the model checking, hence we have a *two stage approach* to state space derivation.

Figure **??** gives an overview of the important classes used by the visualisation part of the plug-in. The most important thing to notice is that there are *two* data structures describing the model. `KModel` is the back-end data structure, used by the abstraction and model checking engines, and stores the Kronecker representation described in Section 2. A `KModel` contains a number of `KComponents`, and each of these contains a number of `RateMatrix` objects — one for each action type in the model. A `RateMatrix` consists of a vector of rates, and a matrix of probabilities. Essentially, it corresponds to a CTMC component, although we have to be a little careful about the mapping between states in the component and indices in the matrix (this is managed by a `KStateSpace` object).

`KDisplayModel` is the front-end data structure, used by the abstraction and model checking views in Eclipse. A `KDisplayModel` object contains a number of components (`KDisplayComponent` objects), which themselves contain a collection of states (`KDisplayState`) and transitions (`KDisplayTransition`) — an implicitly-linked graph data structure, as opposed to a matrix. This makes it easier to map onto a Zest `Graph` object in order to actually draw the graph in the abstraction view.

To understand this separation, it is necessary to explain a little about the structure of the tool. The PEPA Eclipse Plug-in is quite a complicated piece of software, but the most important functionality is separated into two modules (called OSGi bundles) — `pepa`, which contains all the back-end functionality such as parsing, model checking, and CTMC solvers, and `pepa.eclipse.ui`, which contains the front-end Eclipse functionality, including the PEPA editor and the various views. Only certain classes in the `pepa` bundle are made externally visible, to minimise the coupling between different bundles. This means that the `pepa.eclipse.ui` bundle only has access to the classes it actually needs, and is unaware of the internal data structures for the Kronecker representation.

This is different to the standard model-view-controller design pattern, in that the two representations of the model are static. Whenever we modify the PEPA model, we need to parse it again, and this causes a new `KModel` and `KDisplayModel` to be constructed. This is necessary, because a small change in the source file can result in a radically different model. It does *not*, however, mean that all the information from the previous version of the model is lost — information such as labels and the layout of the graphical view are stored by the plug-in in an XML file, which is then re-loaded so that the data can be re-attached to the model as closely as possible.

Atomic properties are managed through a `KDisplayPropertyMap` object, which is associated with a `KDisplayComponent`. A new atomic property is created by a request to the `PropertyBank`. This creates a new `AtomicProperty` object, which records which states (in each component) are labelled with the property[4]. `AtomicProperty` objects are not visible to the abstraction view — they can only be accessed and modified through a `KDisplayPropertyMap`.

Because CSL properties are managed at the level of the entire model (rather than for each component), these are created and modified through `KDisplayModel`. Again, the `PropertyBank` is responsible for storing the properties (which are instances of a subclass of `CSLAbstractStateProperty`), and keeps a link between the atomic properties in the abstract syntax of the CSL property, and the actual atomic properties. The abstraction and model checking views in Eclipse register with the `PropertyBank` (through `KDisplayModel`) as a *listener* for when properties are changed (implementing the `IPropertyChangedListener` interface). This ensures that they are notified of any changes, so that the abstraction and model checking views remain synchronised with one another.

7 Related work

Our visualisation has taken the textual representation of a PEPA model as the primary source in order to be compatible with other modelling and analysis tools which process PEPA models such as IPC [6] and GPA [25]. The PEPA language has enjoyed a wide range of tool support from the PEPA Workbench [13] to PRISM [18] and the PEPA Eclipse Plug-in. The availability of this wide range of modelling and analysis tools influences decisions about matters such as visualisation today.

The PEPA Workbench had a unique feature in that it offered a single-step navigation integrated with a visualisation of the PEPA model. The focus was entirely on behavioural debugging (i.e. finding deadlocks or dead code where actions of the model can never fire). Quantitative information is not represented at all, even at the level of showing rate names on transition labels, much less the values associated to these transitions obtained by evaluating the apparent rate definitions in PEPA. Thus this view does not help modellers who are trying to find why their PEPA model fails to satisfy a probabilistic model-checking formula that they think it should.

Other attempts to add a graphical dimension to PEPA have included the instance of the DrawNET editor which allowed the user to create a PEPA model by editing this graphically [12]. DrawNET provides graphical editors for both the parallel composition

[4] Atomic properties are compositional, which means that a state in the model satisfies an atomic property if and only if each sequential component is in a state that is labelled with the property.

language of PEPA and the sequential component sub-language. DrawNET uses hierarchies of views to represent nested parallel compositions, with sequential components being the leaves of the model tree. Rate information is represented in the automata view in that rate names can be associated with transitions (as well as activity names). However, rate names cannot be evaluated to values, and the apparent rate calculation cannot be performed in the editor.

Solutions which start with a graphical representation of the parallel composition as an arbitrary graph, as DrawNET does, must address the problem that it is possible to draw model configurations which compose components in a way which is not possible to linearise into the standard textual description of PEPA. This is a familiar 2-D to 1-D mapping problem encountered in graph-to-text mapping. We were able to avoid this problem by starting with the textual representation and generating the graphical representation.

The TAPAs modelling tool [7] is a didactic tool for the analysis of process algebra. The idea was to develop a tool that does not implement only a single process algebra, but is instead a framework in which new process algebras can be easily added. At present TAPAs implements CCSP (a process algebra with features of both Milner's CCS and Hoare's CSP) and PEPA. TAPAs avoids the problem of graph-to-text mapping for parallel compositions by allowing only tree-shaped representations of compositions.

8 Conclusions

Both textual and graphical performance modelling formalisms have their advantages and disadvantages, but the use of one does not necessarily have to preclude the other. PEPA is a highly successful language-based approach to performance modelling, and yet there are great benefits from being able to *visualise* a model in a more graphical way. To this end, we have created a novel interface for visualising PEPA models, as part of the PEPA Eclipse Plug-in. The latest version is available for download from http://www.dcs.ed.ac.uk/pepa/tools/plugin.

In future work, there are many additional ways in which we can increase the functionality of the plug-in. One idea would be to allow the model to be edited via the graphical interface, rather than just being viewed passively. This would require some additional data structures to maintain the synchronisation between the editor and the abstraction view, but would lead us more in the direction of a truly combined textual/graphical approach to modelling. Furthermore, there is a great deal of scope for more advanced visualisation, such as animating the *dynamic behaviour* of the model.

To summarise, visualisation is a powerful tool in performance modelling, even for experienced modellers, as it allows a better understanding of the model — particularly in the face of sophisticated transformations such as state-space aggregation and other abstraction techniques. To the best of our knowledge, this is a unique feature of our modelling tool, and we hope that it will be a benefit to the performance modelling community, and provide inspiration for similar features in other modelling tools. Seeing may not always be believing, but it certainly helps in understanding!

Acknowledgements: A prototype implementation of some improvements to the visualisation of PEPA models in the PEPA Eclipse Plug-in was completed by Nan Ai in his Master's thesis [2]. This was very helpful for allowing us to generate and assess different ideas related to visualisation. The implementation of the visualisation, abstraction, and model checking features of the plug-in was supported initially by a Microsoft Research European PhD Scholarship, and subsequently by the Danish Research Council (FTP grant 09-073796). The PEPA case study example included in this paper was created by Allan Clark while working on the EU FET-IST Global Computing 2 project SENSORIA ("Software Engineering for Service-Oriented Overlay Computers" (IST-3-016004-IP-09)).

References

1. The Eclipse platform. `http://www.eclipse.org`.
2. N. Ai. An Enhanced Abstraction View for the PEPA Eclipse Plug-in. Master's thesis, School of Informatics, The University of Edinburgh, 2010.
3. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.
4. G. Balbo. Introduction to stochastic Petri nets. In *Lectures on Formal Methods and Performance Analysis*, pages 84–155. Springer-Verlag, 2002.
5. P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *Proceedings of the 23rd Latin American Conference on Informatics (CLEI 2007)*, 2007.
6. J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 344–351. IEEE Computer Society Press, 2003.
7. F. Calzolai, R. de Nicola, M. Loreti, and F. Tiezzi. TAPAs: A Tool for the Analysis of Process Algebras. In *Transactions on Petri Nets and Other Models of Concurrency I*, pages 54–70. Springer-Verlag, 2008.
8. C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance modelling with UML and stochastic process algebras. *IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, 2003.
9. I. Cappello, A. Clark, S. Gilmore, D. Latella, M. Loreti, P. Quaglia, and S. Schivo. Quantitative analysis of services. In *Rigorous Software Engineering for Service-Oriented Systems*. Springer-Verlag, 2011.
10. A. Clark and S. Gilmore. State-aware performance analysis with eXtended stochastic probes. In *EPEW '08: Proceedings of the 5th European Performance Engineering Workshop on Computer Performance Engineering*, pages 125–140. Springer-Verlag, 2008.
11. D. Daly, D.D. Deavours, J.M. Doyle, A.J. Stillman, P.G. Webster, and W.H. Sanders. Möbius: An extensible framework for performance and dependability modeling. In *Multi-Workshop on Formal Methods in Performance Evaluation and Applications*, 1999.
12. S. Gilmore and M. Gribaudo. Graphical modelling of process algebras with DrawNET. In *Proceedings of the tools appendix to the International multiconference on Measurement, Modelling and Evaluation of Computer-Communication systems*, 2003.
13. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS. Springer-Verlag, 1994.
14. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

15. J. Hillston and L. Kloul. An efficient Kronecker representation for PEPA models. In *Proceedings of the Joint International Workshop, PAPM-PROBMIV '01*, volume 2165 of *LNCS*, pages 120–135. Springer, 2001.
16. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
17. J.-P. Katoen, M. Khattri, and I.S. Zapreevt. A Markov reward model checker. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 243–244. IEEE Computer Society, 2005.
18. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, 2011.
19. A. Movaghar and J.F. Meyer. Performability modelling with stochastic activity networks. In *Proceedings of 1984 Real-Time Symposium*, pages 8–40, 1984.
20. B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Performance Evaluation Review*, 13(2):147–154, 1985.
21. SENSORIA Web site. *SENSORIA: Software engineering for service-oriented overlay computers*. http://www.sensoria-ist.edu, 2011.
22. M.J.A. Smith. Abstraction and model checking in the PEPA plug-in for Eclipse. In *Proceedings of the 7th International Conference on the Quantitative Evaluation of Systems (QEST 2010)*, pages 155–156. IEEE Press, 2010.
23. M.J.A. Smith. Compositional abstraction of PEPA models for transient analysis. In *Proceedings of the 7th European Performance Engineering Workshop (EPEW 2010)*, volume 6342 of *LNCS*, pages 252–267. Springer, 2010.
24. M.J.A. Smith. Compositional abstractions for long-run properties of stochastic systems. In *Proceedings of the 8th International Conference on the Quantitative Evaluation of Systems (QEST 2011)*. IEEE Press, 2011. To appear.
25. A. Stefanek, R.A. Hayden, and J.T. Bradley. GPA — Tool for rapid analysis of very large scale PEPA models. In *Proceedings of the 26th UK Performance Engineering Workshop (UKPEW 2010)*, pages 91–101, 2010.
26. T. Suto, J.T. Bradley, and W.J. Knottenbelt. Performance trees: A new approach to quantitative performance specification. In *MASCOTS'06, 14th International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 303–313. IEEE Computer Society Press, 2006.
27. N. Thomas, M. Munro, P. King, and R. Pooley. Visual representation of stochastic process algebra models. In *Proceedings of the 2nd International Workshop on Software and Performance (WOSP'00)*, pages 18–19. ACM, 2000.
28. M. Tribastone, A. Duguid, and S. Gilmore. The PEPA Eclipse plugin. *SIGMETRICS Performance Evaluation Review*, 36(4):28–33, 2009.