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 allow compositional modelling that is portable and easy to manage. In contrast, 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* [16, 19] 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 [3] and stochastic activity networks [20]. 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 [4] and Möbius [11] — they suffer from some limitations. Most notably, *portability* of the model between tools, and *flexibility* of the tool, since the interface may be too restrictive or cumbersome for advanced users, compared to the freedom of a text editor.

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 [30] that allows performance models in the Performance Evaluation Process Algebra (PEPA) [16] 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 the analysis features of our tool in [24], implementing the compositional abstractions developed in [25, 26], it is important to clarify the purpose of this paper. Our focus here is not on the back-end of the tool, but on the *novel user interfaces* that we have developed. We present some significant improvements in features and usability compared with [24], and moreover describe the implementation details of our front-end, to allow the principles to be applied to other tools based on stochastic process algebras. 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 for labelling states. In Section 5, we then present the interface for constructing performance properties (in the Continuous Stochastic Logic (CSL) [2]), which ensures that the user can only enter syntactically valid properties. We discuss implementation details in Section 6, before considering related work in Section 7 and concluding in Section 8.

2 Modelling in PEPA

The Performance Evaluation Process Algebra (PEPA) [16] 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:

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

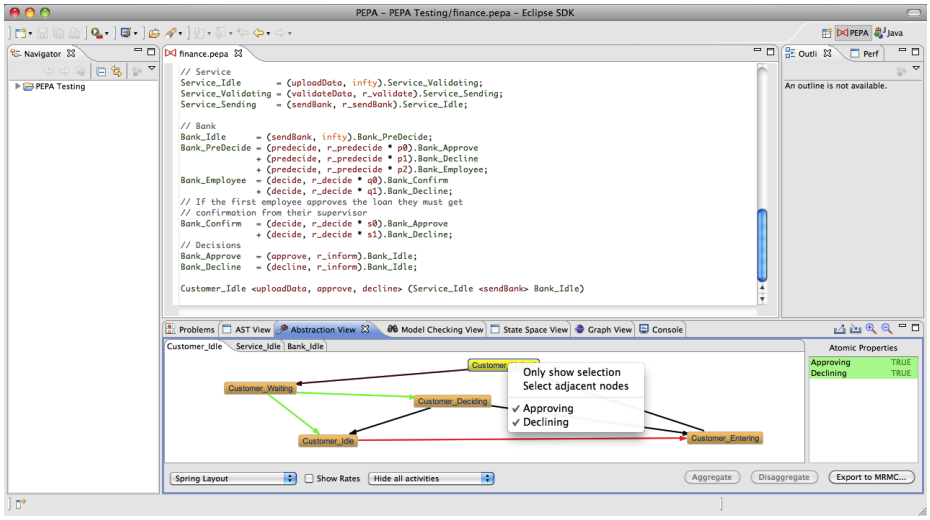

Fig. 1. The PEPA Eclipse Plug-in, showing the editor and the abstraction view

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 (the 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	$A \stackrel{def}{=} C$	The name A refers to component C.

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 [16]. 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 [17], and developed further in [25,26]. It was proven in [25] 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 [16].

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 properties). 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 [21].

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 operator 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 [16] 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:

$$[\![C]\!]_a = [\![C]\!]^{PEPA}_a \quad \text{if } C \text{ is a sequential component}$$

$$[\![C_1 \underset{\mathcal{L}}{\bowtie} C_2]\!]_a = \begin{cases} [\![C_1]\!]_a \circledast [\![C_2]\!]_a & \text{if } a \in \mathcal{L} \\ [\![C_1]\!]_a \odot [\![C_2]\!]_a & \text{if } a \notin \mathcal{L} \end{cases}$$

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

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

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 [23]. 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: the workflow must be reliable, to guard against fraud, and also meet legal constraints on fiscal and monetary transactions.

The PEPA model presented here describes this workflow in terms of a customer using a service portal. Through this, the customer interacts 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, requiring them to co-operate on shared activities (such as $uploadData$) and to proceed independently on other activities (such as the $decide$ activity, which approves the loan request).

Some components have a relatively complex workflow with multi-way branching and loops 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 in this paper for visualising PEPA models differs from the approach taken in graphical modelling formalisms such as Petri nets and Stochastic Activity Networks (SANs), where the visual representation and layout of the model is central. In these formalisms, the modeller most often creates a manual layout of the model — this is the case for the PIPE Petri net editor [4] 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. Some tools for stochastic process algebras — most notably CASPA [22] and Aemilia [6] — have also used this approach to provide a graphical formalism as an alternative to the textual language.

In contrast to this, we wanted the *textual representation* of the PEPA model to be considered as the model source — having primary importance — and for graphical representations to be automatically derived from this and have secondary importance. The idea is to use 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 so that it 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 is important to us that the graphical representation of the PEPA model should be based on the components that 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 using this to structure the visualisation is more comprehensible than looking at the underlying CTMC. In other words, if we show the synchronisation between components by expanding out the model, then the graphical representation becomes too large, and difficult to understand. Finding a good solution to this problem is an important aspect of future work.

Given this component-level perspective, it is important that we visualise both the structural and stochastic information in each component. This means that rates must play an essential part in the visual representation, so that we can detect errors in rate definitions, such as placing the decimal place in an unintended position. Similarly, we should have some way of noticing if we incorrectly declare an activity to be passive rather than active. Our solution is to use different colours to distinguish active and passive activities, and different shades of colour to distinguish fast and slow activities.

4 Visualisation of PEPA Models

The PEPA Eclipse Plug-in allows us to edit and analyse PEPA models using the popular Eclipse framework [12]. This separates the user interface into two main parts. The *editor* 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 invoking analyses of the model.

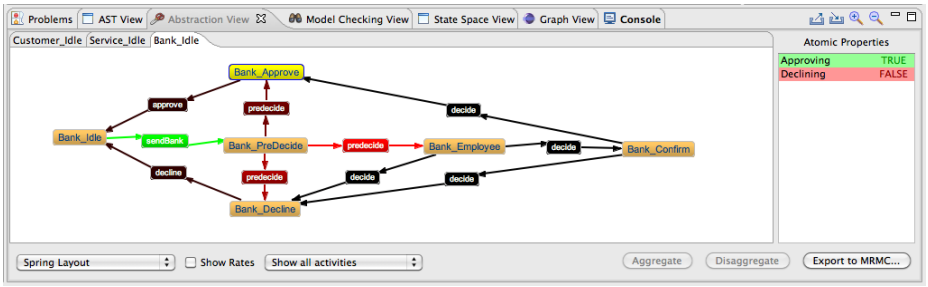

Fig. 3. The abstraction view

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 has a graphical interface for viewing PEPA models, built on top of the Eclipse Graphical Editing Framework (GEF) and the Zest toolkit.

The idea of visualising PEPA models is not a new one, and was first proposed in [29]. 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 [29] 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 Section 2 — 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 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, 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 gives a list of properties that can be selected, or deselected. 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 with that label. We specify atomic properties 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 [25] and stochastic bounds [26] are used to construct an abstract model that is passed to the model checker for analysis. The front-end interface for this 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 of the MRMC model checker [18]. 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 toolkit. 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), but the other nodes remain in their correct position. If the tool detects a new state for which it 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. Sometimes, we want to directly analyse a simple property, such as the steady

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

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. We often want to ask more sophisticated questions, however, 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 use a regular-expression syntax to query the passage time distribution between events, and Continuous Stochastic Logic (CSL) [2], described here.

There are two types of CSL properties: state properties Φ, 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 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 to MRMC.

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

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

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	$\mathcal{S}_{\geq p}(\Phi)$	In the long-run behaviour of the model, does Φ hold with probability at least p?
Next	$\mathcal{P}_{\geq p}(X\ \Phi)$	In the next state, does Φ hold with probability at least p?
Until	$\mathcal{P}_{\geq p}(\Phi_1\ U^{\leq t}\ \Phi_2)$	Will Φ_2 become true no later than time t, and Φ_1 hold at all times until this point, with probability at least p?
Eventually	$\mathcal{P}_{\geq p}(F^{\leq t}\ \Phi)$	Will Φ become true no later than time t with probability at least p?
Globally	$\mathcal{P}_{\geq p}(G^{\leq t}\ \Phi)$	Will Φ always be true until time t with probability at least p?

At the top level, we also support *quantitative* CSL properties, of the form $\mathcal{S}_{=?}(\Phi)$ 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.

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 probability in the original model is between 0.1 and 0.3. If a qualitative property returns 'maybe', or a probability interval is too wide, we should experiment with different abstractions to achieve better results. The theory underlying the tool [25, 26] guarantees the accuracy of the model checker, but the precision depends on the choice of abstraction.

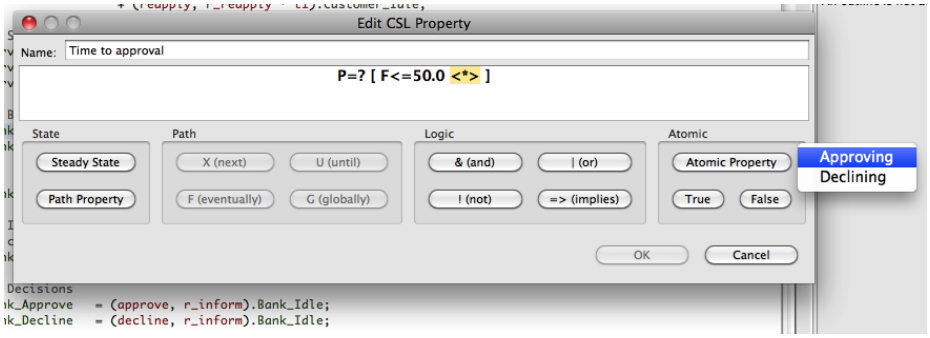

Fig. 4. The CSL property editor

To specify CSL properties and perform model checking, the plug-in provides a **model checking view**. This is basically 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 approaches when it comes to helping a user specify a performance property. One approach is to provide a graphical, user-friendly language for specifying properties, such as performance trees [28]. Another is to have a simple dialog box where the user types in the property in a logic like CSL, such as in PRISM [19]. Our approach lies somewhere in between, 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 of the tool — the interface supports the user by providing them with the correct syntax.

Figure 4 shows the CSL editor, where the property $\mathcal{P}_{=?}(F^{\leq 50}$ "Approving") is being entered for the example in Figure 2 — querying the probability that a loan is approved within 50 time units. 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 (there are keyboard shortcuts for experienced users). 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 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, 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, and displays an error message if we try to do so.

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'.

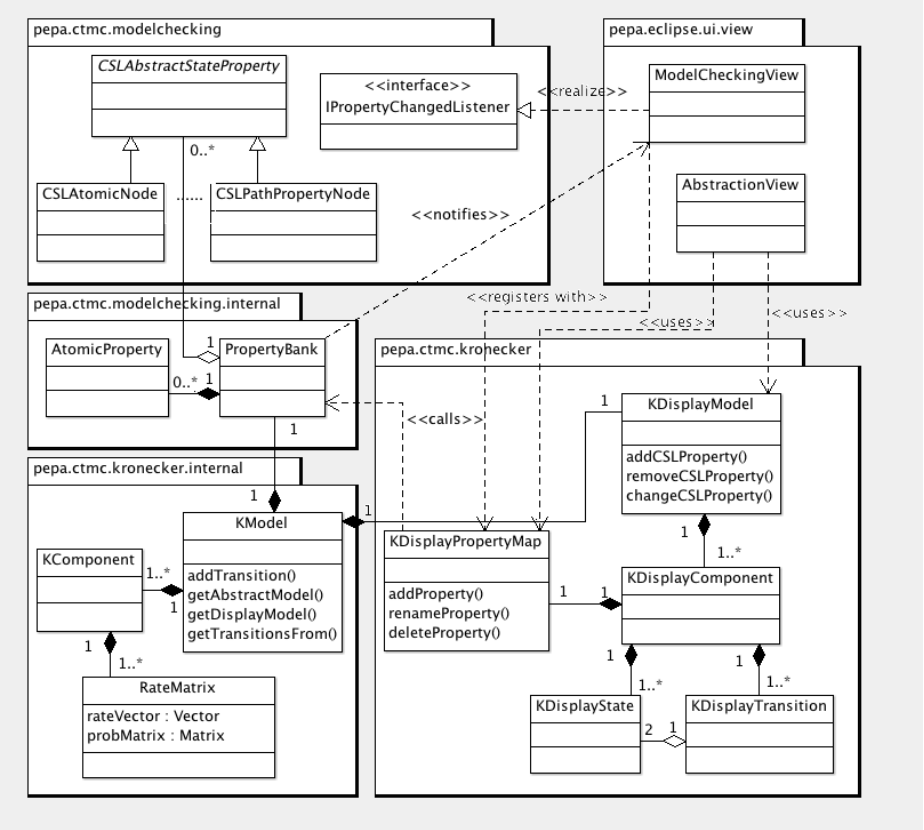

Fig. 5. A UML class diagram of the main classes involved in visualisation in the plug-in

6 Implementation Details

The key idea behind the implementation of the visualisation, abstraction, and model checking functionality of the plug-in is the Kronecker representation described in Section 2. We *partially* derive the state space of the PEPA model when it is parsed, constructing the CTMC component for each sequential component and action type. We only derive the state space of the model when we perform model checking on it — it should be noted that we do this by exploring the reachable state space, and *not* by explicitly performing the matrix operations described in Section 2!

Figure 5 gives an overview of the important classes used by the visualisation part of the plug-in. The most important point is that there are *two* data structures describing the PEPA model. `KModel` is the back-end data structure, used by the abstraction and model checking engines, and stores the Kronecker representation of the model. A `KModel` contains a number of `KComponents`, which in turn contain 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 (omitted from the UML diagram).

`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 contain themselves 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 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 `pepa.eclipse.ui` only has access to the classes it actually needs, and is unaware of the internal data structures for the Kronecker representation.

This separation of concerns differs from 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 creates a new `KModel` and `KDisplayModel`. 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]. This is hidden from the abstraction view, and must be accessed through a `KDisplayPropertyMap`.

Because CSL properties are managed at the level of the entire model (rather than for each component), they 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 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* (implementing `IPropertyChangedListener`). 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 that process

[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.

PEPA models, such as IPC [5] and GPA [27]. The PEPA language has enjoyed a wide range of tool support from the PEPA Workbench [15] to PRISM [19] and the PEPA Eclipse Plug-in, which influences decisions about matters such as visualisation.

The PEPA Workbench contained a single-step navigator, which provided a visualisation of the PEPA model for behavioural debugging (i.e. finding deadlocks or dead code where actions of the model can never fire). This did not show any quantitative information, however, and so did not help modellers to work out why a probabilistic model-checking formula fails to hold, when they think it should.

Other attempts to add a graphical dimension to PEPA have included the DrawNET editor, which allowed the user to create a PEPA model by editing it graphically [14]. DrawNET provides graphical editors for both the parallel composition language of PEPA and the sequential component sub-language. It has also been used to build a graphical interface for the process algebra Aemilia [6].

Other stochastic process algebras also have graphical editors. CASPA [22] uses the Eclipse Graphical Modelling Framework (GMF) [13], which is built on top of GEF, to provide an interface for constructing and editing models. This allows the textual representation of the model to be exported from, and imported into the graphical representation, but the two representations are stored separately. This is in contrast to the PEPA Eclipse Plug-in, where we *only* store the textual representation of the model — generating the graphical representation whenever the model is loaded — which removes the problem of keeping the two representations synchronised. It should be noted that our interface is built on top of GEF directly, and not GMF — this is so that we can make the interface cleaner and simpler, since we do not currently support editing.

A more general modelling tool with a graphical front-end is TAPAs [7]: a didactic tool for the analysis of process algebra. The idea was to develop 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.

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 — for example, by making use of the editing functionality of GMF. This would require us to continually maintain the synchronisation between the editor and the abstraction view, but would lead us in the direction of a truly combined textual/graphical approach to modelling. Furthermore, there is a great deal of scope for advanced visualisation, such as illustrating how components interact with one another, and 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 was completed by Nan Ai in his Master's thesis [1]. 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. N. Ai. An Enhanced Abstraction View for the PEPA Eclipse Plug-in. Master's thesis, School of Informatics, The University of Edinburgh, 2010.
2. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.
3. G. Balbo. Introduction to stochastic Petri nets. In *Lectures on Formal Methods and Performance Analysis*, pages 84–155. Springer-Verlag, 2002.
4. P. Bonet, C.M. Llado, R. Puijaner, and W.J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *23rd Latin American Conference on Informatics*, 2007.
5. 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 Press, 2003.
6. F. Calvarese, A. Di Marco, and I. Malavolta. Building graphical support for Aemilia ADL. Technical Report TRCS 008/2007, University of L'Aquila, 2007.
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 *Proceedings of the 5th European Performance Engineering Workshop (EPEW)*, 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. The Eclipse platform. `http://www.eclipse.org`.
13. The Eclipse Graphical Modeling Framework (GMF). `http://www.eclipse.org/modeling/gmf/`.

14. 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.
15. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368. Springer-Verlag, 1994.
16. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
17. 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.
18. J.-P. Katoen, M. Khattri, and I.S. Zapreevt. A Markov reward model checker. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 243–244. IEEE Press, 2005.
19. 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)*, 2011.
20. A. Movaghar and J.F. Meyer. Performability modelling with stochastic activity networks. In *Proceedings of 1984 Real-Time Symposium*, pages 8–40, 1984.
21. B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *SIGMETRICS Performance Evaluation Review*, 13(2):147–154, 1985.
22. M. Riedl, J. Schuster, and M. Siegle. Recent extensions to the stochastic process algebra tool CASPA. In *Proceedings of the 5th International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 113–114. IEEE Press, 2008.
23. SENSORIA Web site. *SENSORIA: Software engineering for service-oriented overlay computers*. http://www.sensoria-ist.edu, 2011.
24. 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)*, pages 155–156. IEEE Press, 2010.
25. M.J.A. Smith. Compositional abstraction of PEPA models for transient analysis. In *Proceedings of the 7th European Performance Engineering Workshop (EPEW)*, volume 6342 of *LNCS*, pages 252–267. Springer, 2010.
26. 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)*. IEEE Press, 2011.
27. 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)*, pages 91–101, 2010.
28. 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 Press, 2006.
29. 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)*, pages 18–19. ACM, 2000.
30. M. Tribastone, A. Duguid, and S. Gilmore. The PEPA Eclipse plugin. *SIGMETRICS Performance Evaluation Review*, 36(4):28–33, 2009.