The PEPA Eclipse Plug-in

Mirco Tribastone
mtribast@inf.ed.ac.uk

Adam Duguid
a.j.duguid@sms.ed.ac.uk

Stephen Gilmore
stg@inf.ed.ac.uk

Laboratory for Foundations of Computer Science
The University of Edinburgh

ABSTRACT

The PEPA Eclipse Plug-in supports the creation and analysis of performance models from small-scale Markov models to large-scale simulation studies and differential equation systems. Whichever form of analysis is used, models are expressed in a single high-level language for quantitative modelling, Performance Evaluation Process Algebra (PEPA).

1. INTRODUCTION

Performance Evaluation Process Algebra (PEPA) [1] is a concise formal language for high-level quantitative modelling. Applied to modelling problems across the breadth of computer systems performance evaluation and beyond it has users in countries all over the world. Software tools for PEPA provide support for the modelling process from the early stages of model development and debugging right through to automating the experimentation process and culminating in visualisation of numerical results in the form of graphs and charts.

A large part of the appeal of PEPA, leading to its widespread adoption, is the clean definition of simple process structures in the language. As with process algebras in the CSP tradition, a PEPA model is a parallel composition of sequential components, thus all PEPA models are finite state by construction. A simple example of a sequential component is `Process1`, shown below.

```
Process1 = (use, r).Process2;
```

Thus `Process1` performs activity `use` at rate `r` and evolves to `Process2` (which performs other activities, eventually returning to `Process1`). A second copy of this component can be composed in parallel with the first to express unsynchronised concurrent execution, written `Process1 <> Process1`, or equivalently `Process1[2]`. An array of six copies would be `Process1[6]`.

Activities allow one component to co-operate with another. Thus if the `CPU1` component is defined as below

```
CPU1 = (use, r).CPU2;
```

then the co-operation `Process1 <use> CPU1` can perform the `use` activity and evolve to be `Process2 <use> CPU2`, with both components moving on together.

Alternative behaviours are expressed as a choice. For example

```
DISK1 = (read, 0.999 * mu).DISK2
      + (fail, 0.001 * mu).DISK0;
```

describes a disk with rare failures. Papers [2] and [3] provide an up-to-date introduction to PEPA. A complete formal description is available in [1].

A distinctive strength of the PEPA language is that it has two consistent quantitative semantic interpretations. The first interprets a PEPA model as giving rise to a Continuous-Time Markov Chain (CTMC) [1]; the second interprets a PEPA model as giving rise to a system of Ordinary Differential Equations (ODEs) [4]. The ODEs approximate a very large discrete-state system as a continuous-state system.

The benefit which this brings to users is that they can begin developing small-scale models using intuitive discrete-state navigators to investigate model behaviour. When larger and larger arrays of components are used the complexity of the model will grow beyond the limits of discrete-state representation. In this case the modeller can apply the fluid-flow semantics instead, generate a system of ODEs and analyse these. The ODE-based representation does not have the same level of support for state-space navigation but does allow large-scale modelling because the size of the system of differential equations does not depend on the number of copies of each sequential component in the PEPA model.

This paper presents the PEPA Eclipse project (home page available at `http://www.dcs.ed.ac.uk/pepa/tools`), a software tool integrated into the popular Eclipse platform which enables Markovian steady-state analysis, stochastic simulation, and ODE analysis of PEPA models. The remainder of this paper is organised as follows. Section 2 gives an overview of Eclipse and discusses the architecture of the tool. Section 3 introduces *Pepato*, the core Application Programming Interface for the language. Section 4 is concerned with the elements of the user interface. Finally, Section 5 concludes the paper and draws future tool development directions.

2. OVERVIEW

2.1 The Eclipse Framework

Eclipse is a software platform written primarily in Java. Initially developed by IBM, it is now open source and is managed by the Eclipse foundation [5]. Eclipse comprises a run-time environment (Equinox) compliant with the OSGi standard, based on an extensible architecture. A *plug-in* is a software component that adds functionality to Equinox. The Eclipse foundation has developed and made freely available a rich set of plug-ins that deliver an integrated development environment (IDE) for this framework. The IDE itself is extensible through the same plug-in mechanism; for instance, one of the most popular plug-in projects for the Eclipse IDE is JDT, a powerful toolkit for Java development.

The IDE revolves around the notion of *workbench*, which represents the main container of the Eclipse user interface. An Eclipse workspace contains a menu bar, a tool bar, a status bar, and a collection of *editors* and *views*. The two latter components are both used as a presentation layer to some underlying business model; an editor to alter the underlying model of its registered types while a view presents contextual information.

The PEPA Eclipse project comprises contributions to the Eclipse framework for the development and the analysis of PEPA performance models. The project is organised as a set of plug-ins which perform various PEPA-related tasks. As with most medium- to large-sized Eclipse projects, these plug-ins are conveniently packaged into *features*. One advantage is that the plug-ins can be directly installed from within Eclipse through a user-friendly *Update Manager*, which also takes care of dependency resolution.

To install the PEPA Eclipse project the user points the Update Manager to `http://www.dcs.ed.ac.uk/pepa/update/`. A few simple steps presented in a wizard dialogue will guide the user through the process of downloading and installing the necessary components on the Eclipse platform. Seamless upgrade to newer versions of the product is available via the same interface.

2.2 Architecture

The architecture of the PEPA Eclipse project exhibits a loosely-coupled *intra-* and *inter*-component interaction to allow ease of maintainability and accomodate further enhancements of functionality. The project consists of the following six plug-ins: 1) Pepato; 2) Common; 3) Common UI; 4) Eclipse Core; 5) Eclipse Core UI; 6) PEPA Help plug-in.

The *Common* and *Common UI* plug-ins provide necessary support to the other plug-ins of the system, as they encapsulate pieces of commonly-used functionality such as routines for path manipulation, services that handle the progress of long-running tasks, and frameworks for plotting tools. Because of their ancillary nature, they will not be discussed further. A user manual is provided in HTML format through the PEPA Help plug-in as an extension of the Eclipse Help system. This is the standard mechanism of documenting plug-ins in Eclipse, which has two major benefits for the user: 1) The documentation for all the installed plug-ins is located in a central repository, easily accessible from the IDE; 2) The user interface can be enriched with hot-keys and hyperlinks to the relevant pages. In this paper we provide a detailed description of the other plug-ins. The concepts are practically applied to a simple running example, which is shown in Figure 1. The PEPA model is amenable to all the kinds of analysis supported by the plug-in, and here it is presented with the concrete syntax accepted by the tool.

```
/* Rate declarations */
r = 1.0;
s = 4.5;
t = 5.5;
/* Sequential component Process */
Process1 = (use, r).Process2;
Process2 = (think, s).Process1;
/* Sequential component CPU */
CPU1 = (use, r).CPU2;
CPU2 = (reset, t).CPU1;
/* System equation */
Process1[8] <use> CPU1[4]
```

Figure 1: A simple PEPA Model

3. PEPATO

The project is centered around *Pepato*, an application programming interface (API) exposing a library for several PEPA-related core tasks. The root object is the abstract syntax tree of a PEPA model, which can be either generated from a PEPA model file, or created programmatically via the API. The abstract syntax tree is the input for the various forms of analysis available in the library: static analysis, Markovian analysis, simulation, and ODE analysis.

3.1 Static Analysis

Static analysis is concerned with the inspection of the abstract syntax tree for the detection of potential problems in the model description as early as possible in the modelling life cycle. Basic checks include issuing warning messages when process definitions or rate variables are declared but not used, or when there are activities in cooperation sets that are not performed by both of the cooperating sequential components. A routine detects potential deadlocks when an action type in a cooperation set is not performed by a cooperating sequential component. The output of this stage is a list of messages, categorised as *warnings* or *errors*, along with the source code location in which the problem has occurred.

3.2 Markovian Analysis

The first stage of Markovian analysis is the exploration of the model's state space. A strength of Pepato's state space exploration tool is the implementation of the aggregation algorithm presented in [6]. The concrete syntax permits the definition of an array of processes in the form $S[N]$, where S is a PEPA component name and $N \geq 1$. In addition to being a shorthand notation for a parallel composition of N identical sequential components, an array of components may be subjected to aggregation through a user-defined option parameter of the library. If aggregation is turned on, an equivalence relation called *isomorphism* is exploited to reduce the state space size of the underlying Markov chain. Informally, the array of components is represented in a *canonical* form in which the information on the position of a sequential component in the array is lost. For example, let $S_1, S_2, \ldots S_N$ be the local derivatives of some sequential component S and $S[3]$ be the system equation. The states $(S_1 \parallel S_2 \parallel S_N)$, $(S_2 \parallel S_1 \parallel S_N)$, and $(S_N \parallel S_1 \parallel S_2)$ would be represented by the same state in the lumped Markov chain. For instance, the PEPA model in Figure 1 has 4096 states without aggregation; if aggregation is enabled, the state space size is reduced to 45. This aggregation technique still suffers from state space explosion, although to a lesser extent. In our experience with PEPA modelling, it has enabled us to analyse Markov chains which would have been otherwise intractable without aggregation.

The interface of the object representing a state space is transparent to the technique used during state space exploration. This represents an instance of a more common concern throughout the design of the library to ease maintainability and extensibility. In this case we leveraged a well-known software design pattern [7]—the *Abstract Factory* pattern—to seamlessly add new implementations of the state space exploration tool. The currently available version performs in-memory explicit enumeration of the state space and can support Markov chains up to about five million states. (The actual maximum number of states depends on the structure of the model and the memory required to keep the state description vector.) However, this architecture may easily accommodate further enhancements—a out-of-memory disk-based solution to store larger state spaces is ongoing work.

Another situation in which extensibility is desirable is the data structure used to store the labelled generator matrix of the underlying Markov chain. Here, we used the *Adapter* pattern to allow for different data structures to be interchangeably used with different implementations of the state space exploration tool. Enabling this flexibility is of utmost importance to optimise the solution, as some solvers perform more efficiently if the generator matrix is stored in certain forms. Pepato uses the Matrix Toolkit for Java library [8] for the storage of the generator matrix as well as the analysis of

the Markov chain. With similar arguments discussed for the architecture of the state space exploration tool, an Abstract Factory pattern is used to expose the array of solvers available in the library. The current version of Pepato performs steady-state analysis of the underlying Markov process.

The vector holding the steady-state distribution is of little use by itself. One way to reason about the performance of the system under study may be to compute the probability mass of a set of states which satisfy certain conditions. Pepato has the notion of *filters* to define such sets. Filters are grouped into two categories: *state-based* and *transition-based* filters. The former are used to specify conditions which must hold on the local states of the sequential components of the PEPA process. The latter match states according to properties on their incoming or outgoing transitions.

Pepato supports the following state-based filters:

Local State Filter This takes as input a local state of a sequential component S, an integer K, and a relational operator $\circ \in \{<, \leq, =, >, \geq, \neq\}$. It returns the set of states in which the number of sequential components in state S, denoted by $\#S$, satisfies the relation $\#S \circ K$. In the sample PEPA model, a performance metric of interest could be the probability of finding all the CPUs in their idle state CPU_2. This can be simply queried by using the local state filter $CPU_2 = 4$.

Pattern Matching Filter A more expressive way of filtering based on local states is available through a pattern-matching filter. For example, the expression $S_i| * |S_j$ matches states that have the first sequential component in state S_i and the third sequential component in state S_j. The wildcard operator $*$ is used to indicate any local state in a position. The same query as above is represented by the following expression:

```
* | * | * | * | * | * | * | CPU2 | CPU2 | CPU2 | CPU2
```

Unnamed State Filter Consider the definition

$$S \stackrel{def}{=} (\alpha, r).(\beta, s).S'$$

The local state $(\beta, s).S'$ is called *unnamed* because it is not defined through a constant. The modeller may want to use unnamed local states because their behaviour is of secondary importance for the performance analysis. This filter returns the set of states in which all its sequential components are not in an unnamed state.

Probability Threshold Filter This filter may be applied to match states whose steady-state probability is above or below a given threshold.

A transition-based filter takes as input an action type and the direction of the transition (i.e. incoming or outgoing). It filters states which have transitions of the given direction labelled with the given action type. Both kinds of filter can be combined using boolean operators.

Another useful way of extracting performance indices of interest from a PEPA model is the application of reward structures to the steady-state probability distribution. A reward function $f : S \to \mathbb{R}$ is a real-valued function on the states of the underlying Markov chain. Let π be the steady-state probability distribution of the Markov chain and i be the index over the state space. The reward is computed as follows

$$R = \sum_i \pi_i \cdot f(S_i)$$

Pepato can be used to compute the following reward structures: throughput, utilisation and population levels. Throughput is associated with an action type and represents the number of activities of a particular type that are performed in a unit of time. Let O_S be the set of outgoing transitions of state S. Let (α, r, S') be a triple representing one such transition. The reward structure for the throughput $T_{\overline{\alpha}}$ of an action $\overline{\alpha}$ is defined as follows

$$T_{\overline{\alpha}} = \sum_{(\alpha,r,S') \in O_S \wedge \alpha = \overline{\alpha}} r$$

In performance evaluation, the notion of utilisation is traditionally associated with queueing networks to indicate the occupancy of a server. In PEPA utilisation is interpreted differently. Consider a sequential component S evolving through a set of local states $\{S_1, S_2, \ldots S_N\}$. The utilisation $U_S(S_i)$ of a local state S_i of a sequential component is the fraction of its lifetime that the sequential component spends in state S_i. Clearly, for all sequential components it must hold that $\sum_i U(S_i) = 1$.

Finally, population levels indicate the average number of copies of a sequential component in the steady state. This reward structure will be used later in this paper when we discuss support for ODE analysis.

On a side note, it is worth mentioning the interoperability capability of Pepato. The API provides support for exporting the transition system of the Markov chain in a row-column-value text format, which can be easily imported by external matrix toolkits such as Matlab. The state descriptor may also be exported in a comma-separated values text format, for processing with tools such as `grep` or `sed`. The state space interface also has an import option to load a steady-state distribution vector computed by third-party solvers.

3.2.1 Stochastic Simulation

The Pepato library also supports time-series analysis by stochastic simulation or solving ODEs through a customised library [9]. These techniques allow the observation of a system as it evolves from an initial state over a period of time, a situation more commonly known as the Initial Value Problem (IVP). In this case the system equation of a PEPA model provides the IVP. More importantly, both techniques can analyse systems that are intractable by Markovian analysis.

This advantage is not simply the result of a more powerful technique, rather it is the combination of these types of analysis when used on a model in the aggregated form. With the Markovian analysis altering the number of copies of an aggregated component impacts the size of the state space, whereas altering the number of activities enabled by a component affects the number of links in the space. The dominant influence on Markovian analysis is the number of copies of each component. The stochastic simulation and ODE solvers however are mostly insensitive to changes in levels of components, being more affected by the number of unique components and activities in the entire system.

The stochastic simulators which are currently supported are Gillespie's Stochastic Simulation Algorithm (SSA) [10] and the Gibson-Bruck algorithm [11]. Originally designed for modelling chemical reactions, they are derived from the same fundamental laws as the master equation, which itself is a Markov chain. It is this structure, and the optimisations possible because of it, which allow these algorithms to analyse in a realistic time systems which would be intractable when using a state-based representation.

The format required by the simulators presents the model in terms of activities, rather than components, with each entry defining a unique firing of that action. Let α be an activity which is enabled in two or more components, in this instance C_1 and C_2.

If these two components do not co-operate over α then they must appear as two unique entries within the simulator. The correct rate at which each unique entry fires is also required, and this even depend on components which are not directly involved in this activity. To generate the correct data the Pepato library uses an intermediate representation to build partial forms as the system equation of the model is traversed.

Evaluation by stochastic simulation will always give good agreement when compared to Markovian analysis as both are stochastic and discrete in nature. The trade-off is computation time. To discover the steady-state of the system from stochastic simulation a number of independent replications must be performed, with a simulation time long enough for the average to tend towards the steady-state. This often means that when Markovian analysis is tractable it will also be the faster method.

3.3 ODE Analysis

Generating the ODEs of a PEPA model requires the same information as the stochastic simulators, but in a different form. Using the intermediate state already created by the Pepato library it is a simple matter to generate the correct ODEs, where each equation is the summation of rates of activities where the component is a derivative and the subtraction of the summation of rates of activities where the component is enabled, also referred to as the exit and entry activities of a component [4]. Figure 2 shows the ODEs generated from the example PEPA model given in Figure 1.

$$\frac{dProcess_1}{dt} = sProcess_2 - r\,\min(Process_1, CPU_1)$$
$$\frac{dProcess_2}{dt} = r\,\min(Process_1, CPU_1) - sProcess_2$$
$$\frac{dCPU_1}{dt} = tCPU_2 - r\,\min(Process_1, CPU_1)$$
$$\frac{dCPU_2}{dt} = r\,\min(Process_1, CPU_1) - tCPU_2$$

Figure 2: The set of ODEs derived from the simple PEPA model in Figure 1.

The Pepato library currently supports two numerical integrators for ODES, a Runge-Kutta implicit-explicit implementation [12] and the Dormand-Prince adaptive step-size solver [13], both through the odeToJava library [14]. In contrast to the stochastic simulators, the ODE solvers are continuous and deterministic, and ill-equipped to deal with discontinuous functions because of this. This is a potential issue when dealing with passive rates within a model, where the rate is not dependent on the population level of a component, but merely its presence. This does not negate the usefulness of the ODE solvers, which are complimentary to the stochastic simulators—as the population sizes within a model grow the confidence in the ODE solution increases, as does the performance gain over the stochastic simulation. An example of the results possible from both families of methods can be seen in Figure 3, where the number of independent replications has been kept low to highlight the stochastic behaviour.

4. THE GRAPHICAL USER INTERFACE

Pepato is available to Eclipse users through the plug-ins *Eclipse Core* and *Eclipse UI*. When developing for Eclipse, it is a recommended practice to separate out core functionality of a service

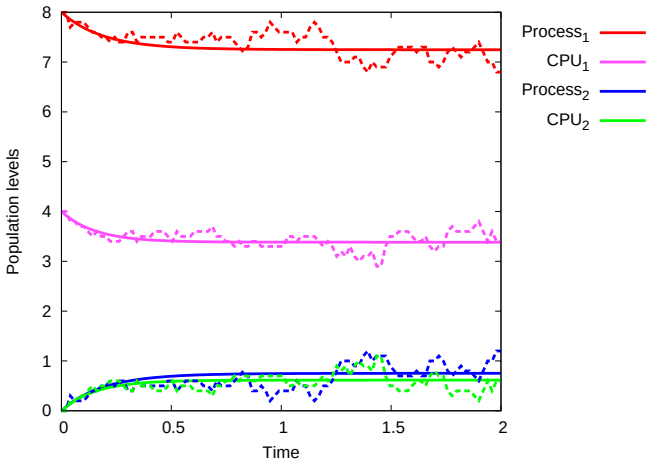

Figure 3: ODE and SSA analysis of the simple PEPA model in Figure 1. Solid lines represent the ODE solution and dotted lines the SSA (performed with 10 independent replications). The legend is ordered by final population level from the ODE solution.

and its contributions to the user interface into (at least) two distinct plug-ins. This allows the core functionality to be used in a context in which the user interface is not necessary or even not available. Eclipse Core exposes Pepato to the platform, and its main role is to provide a mapping between files managed within Eclipse and PEPA-related objects. In particular, it is based on the Eclipse *Resources* plug-in, which implements a file-system layer for the Eclipse workbench (i.e., the *workspace*) on top of the native file system of the underlying operating system. This facilitates the management of events related to changes in the state of workspace files. For example, *listeners* may be installed on files to be notified when a file being edited is saved. Eclipse Core registers listeners to PEPA model files, which trigger the automatic execution of the PEPA parser and the static analysis routines when the model is saved. Eclipse Core also represents a good entry point for developers of third-party plug-ins to use PEPA-related services.

Eclipse UI contains all the user interface contributions to the Eclipse IDE. It features an editor, which is automatically associated by the workbench to workspace files with the `.pepa` extension. The editor has syntax highlighting and supports graphical annotations (*markers*) for problems encountered during the modelling process. Tasks to be performed on the PEPA model being edited are shown in the top-level menu bar, and a number of views are connected to the editor. A customisable arrangement of all the views of interest to a PEPA modeller is provided in the PEPA *perspective*. Figure 4 is a screen-shot of the Eclipse workbench with a possible layout for modelling with PEPA. In this section we provide a detailed discussion of the views and the actions available under the Eclipse UI PEPA plug-in.

4.1 Contributions to Other Plug-ins

The *Navigator* view is used to navigate the Eclipse workspace. Workspace files with the `.pepa` extension are associated with the PEPA synchronisation icon in the editor and registered with the PEPA editor. The *Problems* view is populated automatically with syntax error and static analysis messages. The plug-in defines two levels of severity: a *warning* allows the user to continue the analysis, whereas an *error* must be fixed. The *Console* view provides

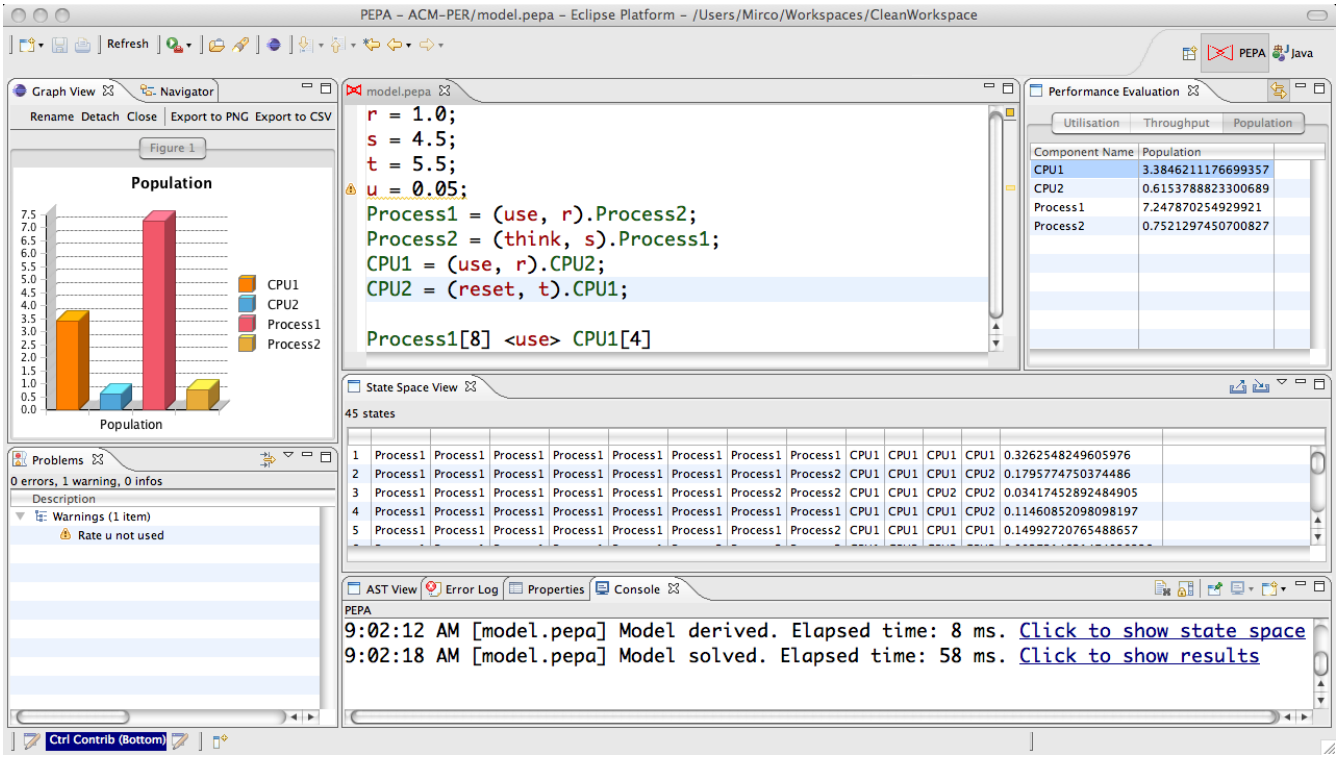

Figure 4: The PEPA Eclipse Plug-in at a glance. This screen-shot shows an instance of the Eclipse 3.3 IDE running on Mac OS X. The top-middle area shows an editor for the simple PEPA model. The model contains a deliberate unused definition (rate u), underlined in the editor area and reported as a problem in the *Problems* view. Below the editor is the *State Space View*, which lists the state space of the Markov chain. The model has been solved for steady-state analysis (as reported in the *Console* view) and the probability distribution is shown as a column in the State Space View. The *Population* tab of the *Performance Evaluation* view is open in the top-right area of the workbench. This data is presented graphically as a bar chart in the *Graph View* (top left).

verbose information on the status of a PEPA model. In particular, it displays execution times of the various stages of analysis and provides hyperlinks which open the related views.

4.2 AST View

The AST (Abstract Syntax Tree) view is connected to the active PEPA editor in the workspace and shows a tree-based graphical representation of the abstract syntax tree of the PEPA model, along with the source code location information as gathered during the scanning and the parsing of the document. It mainly serves debugging purposes and is particularly useful for developers who wish to manipulate PEPA abstract syntax trees programmatically.

4.3 State Space View

The State Space View is linked to the active PEPA editor and provides a tabular representation of the state space of the underlying Markov chain. The table is populated automatically when the state space exploration is invoked from the corresponding top-level menu item. A row represents a state of the Markov chain, each cell in the table showing the local state of a sequential component. The order in which sequential components are displayed corresponds to the order in which they are found in the cooperation set by depth-first visit of the cooperation's binary tree. A further column displays the steady-state probability distribution if one is available.

A toolbar menu item provides access to the user interface for managing state space filters. When a set of filter rules is activated, the excluded states are removed from the table. The probability mass of the states that match the filters is automatically computed and shown in the view. Filter rules are assigned names and made persistent across workspace sessions. From the toolbar the user can invoke a wizard dialogue box to export the transition system and one to import the steady-state probability distribution as computed by external tools.

The view also has a *Single-step Debugger*, a tool for navigating the transition system of the Markov chain. The debugger can be opened from any state of the chain and its layout is as follows. In an external window are displayed the state description of the current state and two tables. The tables show the set of states for which there is a transition to or from the current state. The tables are laid out similarly to the view's main table. In addition, the action types that label a transition are shown in a further column. The user can navigate backwards and forwards by selecting any of the states listed.

4.4 Performance Evaluation View and Graph View

A wizard dialogue box accessible from the top-level menu bar guides the user through the process of performing steady-state analysis on the Markov chain. The user can choose between an array of iterative solvers and tune their parameters as needed. Performance metrics are calculated automatically and displayed in the Performance Evaluation View. It has three tabs showing the results of the aforementioned reward structures (throughput, utilisation, and pop-

ulation levels). Throughput and population levels are arranged in a tabular fashion, whereas utilisation is shown in a two-level tree. Each top-level node corresponds to a sequential component and its children are its local states.

The Performance Evaluation View can feed input to the Graph View, a general-purpose view available in the plug-in for visualising charts. Throughputs and population levels are shown as bar charts and a top-level node of the utilisation tree is shown as a pie chart. As with any kind of graph displayed in the view, a number of converting options is available. The graph can be exported to PDF or SVG and the underlying data can be extracted into a comma-separated value text file.

4.5 Experimenting with Markovian Analysis

An important stage in performance modelling is sensitivity analysis, i.e. the study of the impact that certain parameters have on the performance of the system. A wizard dialogue box is available in the plug-in to assist the user with the set-up of sensitivity analysis experiments over the models. The parameters that can be subjected to this analysis are the rate definitions and number of replications of the array of processes in the system equation. The performance metrics that can be analysed are throughput, utilisation, or population levels. If the model has filter rules defined, the probability mass of the set of filtered states can be used as a performance index as well. The tool allows the set-up of multiple experiments of two kinds: one-dimensional (performance metric vs. one parameter) or two-dimensional (performance metric vs. two parameters changed simultaneously). The results of the analysis are shown in the Graph View as line charts.

For example, a parameter that may have an important impact on the performance of the system is the reset delay of the CPU.

4.6 Time-Series Analysis

When performing a time-series analysis there are three basic steps to complete; component selection, solver selection and solver parameterization, all of which are handled by the time-series analysis wizard. Rather than simply observing all components, the wizard allows the modeller to select only those components that are of interest. This becomes more pertinent as either the number components in the system or number of observed time points increase—one limitation of the current time-series solvers is that all data is held in memory, and only written out to disk when exporting from the graph view. Solver selection and parameterization are self-explanatory, with the list of visible parameters being dynamically linked to the currently selected solver.

In keeping with the rest of the UI, the selections across all three steps are persistent across invocations. Likewise, each unique parameter is stored only once, meaning parameters such as start and stop times are persistent over all solvers. Lastly, the parameters, including selected solver, are attached to the results in the graph view for future reference. Currently this meta data can only be seen when the data is exported. The last feature of the wizard is the ability to export the model in alternative formats, such as Matlab.

5. CONCLUSION

In this paper we presented the PEPA Eclipse Plug-in Project, a toolkit to support the timed process algebra PEPA for the Eclipse framework. The project allows the user to perform steady-state Markovian analysis, stochastic simulation, and fluid-flow analysis of PEPA models. Although one of the main points of strength is the integration with the Eclipse platform and its development environment, the Pepato library can also be used by third-party Java-based applications. For instance, in the analysis of large-sized models the user-friendliness of the graphical interface may be desirably traded with a less demanding interface. We are currently working on the implementation of a command-line interface for Pepato.

The plug-in project is under active development, and a number of extensions have been planned for future work. With regards to Markovian analysis, we are developing software modules for transient analysis and the computation of response-time quantiles. The experimentation framework, currently available for Markovian analysis only, is being extended to ODE analysis and stochastic simulation.

Acknowledgement

The helpful comments of the anonymous referees are appreciated. This work has been partially sponsored by the EU-funded project SENSORIA, IST-2005-016004.

6. REFERENCES

[1] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[2] J. Hillston. Tuning systems: From composition to performance. *The Computer Journal*, 48(4):385–400, May 2005. The Needham Lecture paper.

[3] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 239–248, Chicago, June 2005. IEEE Computer Society Press.

[4] J. Hillston. Fluid flow approximation of PEPA models. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 33–43, Torino, Italy, September 2005. IEEE Computer Society Press.

[5] Eclipse Foundation. Eclipse home page. `http://eclipse.org`.

[6] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, May 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] B.-O. Heimsund. MTJ: Matrix Toolkit for Java. `http://ressim.berlios.de/`.

[9] CompBio Group, Institute for Systems Biology. ISBJava. Available at `http://magnet.systemsbiology.net/software/ISBJava/`.

[10] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[11] M.A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry*, 104:1876–1889, 2000.

[12] U. M. Ascher, S. Ruuth, and R. Spiteri. Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, November 1997.

[13] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, March 1980.

[14] odeToJava library. Available at `http://www.netlib.org/ode/odeToJava.tgz`.