11 The PEPA Plug-in for Eclipse

In this lecture note we introduce the tool support which is available when modelling with PEPA. Undertaking modelling studies of any reasonable size is only possible if the modelling process has adequate support. PEPA is fortunate to be supported by a number of different tools developed in Edinburgh and elsewhere. However in this note we will focus on the PEPA Plug-in for Eclipse as this is the tool you will be using for your practical.

11.1 Representing PEPA in ASCII

As mentioned in the previous lecture note, PEPA is often presented as a formal language which incorporates some distinctive characters, particularly the cooperation operator $\underset{L}{\bowtie}$, produced using the typesetting package LaTeX. When we are writing a `.pepa` file for processing by one of the tools we are restricted to use the standard ASCII character set, so some modification of representation is necessary. The small table below shows how to represent PEPA in ASCII.

Combinator	LaTeX representation	ASCII representation
Prefix	$(a, r).P$	`(a, r).P`
Choice	$P + Q$	`P + Q`
Cooperation	$P \underset{\{a\}}{\bowtie} Q$	`P < a > Q`
Parallel	$P \parallel Q$	`P < > Q`
Constant	$A \stackrel{def}{=} (a, r).P$	`A = (a, r).P;`

Another difference from the formal representation, is that in the ASCII representation each component definition must be terminated by a "`;`". The rates associated with activities, may be given concrete values within the definition, e.g. `(a, 6)`, or may be expressed in terms of variables, e.g. `(a, r)`. In this latter case the variable must be assigned a value at the top of the `.pepa` file, e.g. `r = 6;`. We saw expressions like this in the example PEPA model (the web service example) presented at the end of Lecture Note 10. You will find several example `.pepa` files on the course webpage.

11.2 The PEPA Plug-in Project

The PEPA Plug-in Project is a software tool to support Markovian analysis of PEPA models. The tool is implemented as a collection of plug-ins for Eclipse, an extensible integrated development environment for a large variety of programming and modelling languages such as Java, C++, Python and UML. This framework was chosen for three main reasons. First, Eclipse is a freely available product. Second, it is widely supported by a growing community of users and businesses. Third, it can run on a variety of platforms, as it is implemented in Java and the graphical library used for the user interface is available on many operating systems.

The functionalities of the tool are accessible both programmatically and through a more user-friendly graphical interface. In this lecture note we focus on the latter method.

Resources of an Eclipse workspace can be manipulated using two main classes of tools, *editors* and *views*. The former follow the traditional open-save-close cycle pattern. The latter are typically used to navigate resources, modify properties of a resource and provide additional information on the resource being edited.

The PEPA Plug-in contributes an editor for the language and views which assist the user during the cycle of model development. Static analysis is used for checking the well-formedness of a model and detecting potential errors prior to inferring the derivation graph of the system. A well-formed model can be derived, i.e. the underlying Markov process is extracted and the corresponding state space can thus be navigated and filtered via the **State Space** view. Finally, the **CTMC** view allows numerical steady-state analyses such as activity throughput and component utilisation.

11.2.1 Editor

A PEPA editor is opened for files in the Eclipse workspace which have the `pepa` extension. The editor provides a convenient way to run a parser which translates the model description in the PEPA language into an in-memory representation suitable for further processing. This form is represented graphically in the **AST** view by means of a hierarchical structure for the model. The in-memory model also acts as an intermediate form for converting PEPA models into external formats.

11.2.2 Static Analysis

Static analysis deals with checking the well-formedness of a PEPA model. Because of its low computational cost, static analysis is performed every time the text of the model description is saved. The output of this tool is a list of messages to the already existing Eclipse **Problem** view. The information provided can be grouped into two categories: *warnings* are messages about low priority problems which do not prevent further processing; *errors* are instead critical problems which must be fixed in order to continue the model development process. For instance, basic warning messages are about rate or process definitions which are defined in the model description but never used; error messages can be about rate or process names which are used but never defined.[1]

More advanced static analysis is carried out to detect potential local deadlocks, redundant declaration of actions of the cooperation operator and unguarded component uses giving rise to non-well-founded definitions of processes, i.e. self-containing processes.

Let us consider two processes which cooperate over a non-empty set of action types. A local deadlock is a condition that may occur when one process cannot proceed because it is in a state where it is synchronised on an activity which can never be performed by its partner. The model in Figure 1 exhibits a local deadlock in the initial state, because the action type α cannot be performed by either *Q1* or *Q2*. Local deadlock conditions are critical errors which can be statically detected by examining the used definition set of each cooperation of a model.

The tool emits warning messages if it discovers the existence of redundant definition of action types in cooperation sets.

[1]It is worthwhile noting that rate names must be declared before using them in a prefix definition. However, there is no such a rule with regards to process definitions.

$$
\begin{aligned}
P1 &\stackrel{def}{=} (\alpha, r).P2 \\
P2 &\stackrel{def}{=} (\gamma, t).P1 \\
Q1 &\stackrel{def}{=} (\beta, s).Q2 \\
Q2 &\stackrel{def}{=} (\epsilon, v).Q1
\end{aligned}
$$

$$P1 \underset{\{\alpha\}}{\bowtie} Q1$$

Figure 27: Example of a PEPA model with local deadlock

$$
\begin{aligned}
P1 &\stackrel{def}{=} (send_{succ}, r).P2 + (send_{fail}, s).P3 \\
P2 &\stackrel{def}{=} (new\text{-}data, t).P1 \\
P3 &\stackrel{def}{=} (reset, u).P1 \\
Q1 &\stackrel{def}{=} (send_{succ}, \top).Q2 + (send_{fail}, \top).Q3 \\
Q2 &\stackrel{def}{=} (file\text{-}data, v).Q1 \\
Q3 &\stackrel{def}{=} (error, w).Q1
\end{aligned}
$$

$$P1 \underset{\{send_{succ}, send_{fail}\}}{\bowtie} Q1$$

Figure 28: ExamplePEPA model.

11.2.3 State Space Derivation

The PEPA Plug-in project provides a tool for state space derivation, i.e. the process of extracting a Markov process from the labeled transition system of the PEPA model. The output of the tool is the state space and the corresponding infinitesimal generator of the Markov process. The state space can be navigated and filtered via the **State Space** view. The state space is represented in a tabular form: the first column is the state number; then follow as many columns as the number of top-level components of the system. The tabular representation of the state space of the model in Figure 2 would be as in Table 1.

A variety of filter options is available in order to narrow down the number of states shown in the view. The user can exclude/include states which have a sequential component in a particular local state or states which contain unnamed processes (i.e., prefixes). More precise filtering can be obtained by means of a pattern language which allows the user to match local states which contain given top-level component local states at specified positions. The components are separated by vertical bars and a wild-card is used to disregard positions which are of no interest. According to the example in Figure 2, the pattern `P1 | *` would match the states whose first top-level component is in state P1, thus displaying states 1,4,7; the pattern `* | Q2` would match states 5,7. For a more

Table 3: Tabular representation of the state space of the example model

State Number	First Component	Second Component
1	*P1*	*Q1*
2	*P3*	*Q3*
3	*P3*	*Q1*
4	*P1*	*Q3*
5	*P2*	*Q2*
6	*P2*	*Q1*
7	*P1*	*Q2*

concise description of the filter, the generic pattern `P` is considered as an abbreviation of `P | * | ... | *`.

Additionally, the plug-in provides the **Single Step Navigator**, a tool for walking over the state space. This is particularly useful for debugging purposes. It consists of two tables containing the list of incoming and outgoing states. The sequential components which cause the transition to be performed are highlighted and an option allows the user to make filtered states not walkable.

11.2.4 Steady-state Analysis

A model whose state space is derived successfully is amenable to performance analysis which can be carried out by calculating the steady-state probability distribution of the CTMC over the state space. The user interface provides a dialogue wizard which guides the user through this process. The wizard is a graphical interface to the MTJ toolkit, the library used for the numerical solution of the Markov chain, allowing the user to choose and tune the parameters of an extensive selection of solvers and preconditioners.

After the model is successfully solved, the State Space view is updated with information on the obtained steady-state probability distribution which is shown on an additional column. Additional analysis can be carried out via the **Performance Evaluation** view, which permits throughput and utilisation analysis.

In order to better illustrate these metrics, let us consider the model in Figure 3 consisting of one single sequential component evolving through three local states. With rates $r = 2$, $s = 1$, $t = 1$ the utilisation figures are $P1 = 0.2, P2 = 0.4, P3 = 0.4$ whereas throughput is 0.4 for each action.

$$\begin{aligned} P1 &\stackrel{def}{=} (start, r).P2 \\ P2 &\stackrel{def}{=} (run, s).P3 \\ P3 &\stackrel{def}{=} (stop, t).P1 \end{aligned}$$

Figure 29: A tiny PEPA model with one sequential component