

A Revised PEPA Probe Implementation

Allan Clark
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract

This paper reports on a revised implementation for the addition of performance measurement probe components. The probe components are specified via a high level description syntax and translated into `pepa` components which are then synchronised with (a portion of) the PEPA model over the activities that the probe seeks to observe. This revised implementation forms the basis of the `pepaprobe` utility which is developed as part of the `ipclib`[1] PEPA library.

1 Introduction

The `pepaprobe`[2] utility takes as input a PEPA[3] model and a number of probe descriptions. These probes are performance measurement specification probes in the style of [4] and describe, using a high-level regular-expression like language, additional components to be combined with the input model. Such probes are intended to be observational and therefore do not change the original behaviour of the given model. The output is the input model transformed according to all of the given probes. This output model can then be solved using existing `pepa` software tools such as [5, 6, 7]

The probe descriptions are translated into `pepa` components which then synchronise with the model (or a part of it). Information about the state at a given time t can be derived by interrogating the state of the probe at time t . The probe components may perform high-priority immediate communication between each other to signal the occurrence of important events.

2 Probe Specification

The major addition of this revised implementation is that labels which are attached to parts of the probe specification have been generalised. In this section labels are briefly described and then some example probes are given. Section 3 gives the full grammar for specifying performance measurement probes.

Labels may be attached to a probe to signify some important event or sequence of events. A label may be attached to the end of any part of a probe. Generally labels are used as communication between multiple probes. In addition there are two specially regarded labels *start* and *stop*. These labels indicate the entering and exiting of the state of interest. If the measurement is a passage-time query then the *start* and *stop* labels indicate the entering and exiting of

the passage to be measured. If used in a steady-state query then the *start* and *stop* labels indicate the entering and exiting of the set of measured states.

The following probe will 'start' on any of the actions *a*, *b* or *c* and, following that will 'stop' on any of the actions *x*, *y* or *z*. Notice that all probes are cyclic and loop back to the start once they have completed.

$$(a \mid b \mid c) : start, (x \mid y \mid z) : stop$$

The following probe waits until it has observed three *a* actions without observing a *b* action. Once this occurs the probe is 'started', to stop the probe it must observe the sequence *c, d*.

$$(a, a, a)/b : start, (c, d) : stop$$

Probes may be attached to a particular component within the model using the double colon syntax. The following probe may be used to ask what is the probability that the *Client* component has performed three or more requests and has yet to see any responses.

$$Client :: (request, request, request)/reponse : start, response : stop$$

3 Probe Grammar

The full grammar for probe specification is given in Figure 1

<i>probe</i>	$:=$	<i>location</i> :: <i>R</i>	A local probe
		<i>R</i>	A global probe
<i>location</i>	$:=$	<i>processId</i>	Attach to a single process
		<i>processId</i> [<i>n</i>]	Attach to an array of processes
<i>R</i>	$:=$	<i>action</i>	Observe an action
		<i>R</i> : <i>label</i>	Send a signal on matching <i>R</i>
		<i>R</i> ₁ , <i>R</i> ₂	<i>R</i> ₁ followed by <i>R</i> ₂
		<i>R</i> ₁ <i>R</i> ₂	<i>R</i> ₁ or <i>R</i> ₂
		<i>R</i> *	zero or more <i>R</i>
		<i>R</i> ⁺	one or more <i>R</i>
		<i>R</i> { <i>n</i> }	<i>n</i> <i>R</i> sequences
		<i>R</i> { <i>m</i> , <i>n</i> }	between <i>m</i> and <i>n</i> <i>R</i> sequences
		<i>R</i> ?	one or zero <i>R</i>
		<i>R</i> / <i>a</i>	<i>R</i> without observing an <i>a</i>

Figure 1: The grammar for probe specification in pepaprobe.

4 Inner workings

A concise algorithm for compiling probes to PEPA was given in [4]. However this algorithm is known to suffer from a problem relating to non-determinism in the probes. Because the probe specification language is a regular expression like language there is some non-determinism involved. Therefore an algorithm

which directly translates the probe specification into PEPA components will suffer from non-determinism.

For example the simple probe:

$$(a, b)|(a, c)$$

Which should allow the probe to observe either the sequence a, b or the sequence a, c , can naïvely be translated into the PEPA components:

$$\begin{aligned} \textit{Probe} &\stackrel{\textit{def}}{=} \textit{ProbeAB} + \textit{ProbeAC} \\ \textit{ProbeAB} &\stackrel{\textit{def}}{=} (a, \top).(b, \top).\textit{Probe} \\ \textit{ProbeAC} &\stackrel{\textit{def}}{=} (a, \top).(c, \top).\textit{Probe} \end{aligned}$$

However this means that when the probe first observes an a activity being performed by the model it must decide which branch of the probe to take. It may of course choose incorrectly. Therefore this probe will perform incorrectly.

To compile this probe correctly there must be a state which indicates that an a activity as been observed and either a b or a c activity is still possible.

$$\begin{aligned} \textit{Probe} &\stackrel{\textit{def}}{=} (a, \textit{top}).\textit{ProbeA} \\ \textit{ProbeA} &\stackrel{\textit{def}}{=} \textit{ProbeB} + \textit{ProbeC} \\ \textit{ProbeB} &\stackrel{\textit{def}}{=} (b, \top).\textit{Probe} \\ \textit{ProbeC} &\stackrel{\textit{def}}{=} (c, \top).\textit{Probe} \end{aligned}$$

4.1 Adding the Self-Loops

Because the probe must be observational only, the probe must be capable of performing all of the actions over which it cooperates with the model at all possible stages. To do this at each state we add a set of self-loops. For each state, any actions in the cooperation set which the translated probe cannot already perform, a self-loop action which loops to the same state is added. In our simple example the start state must be able to loop on actions b and c while the holding state must be able to loop on the a action. Here are the updated definitions:

$$\begin{aligned} \textit{Probe} &\stackrel{\textit{def}}{=} (a, \textit{top}).\textit{ProbeA} \\ &\quad + (b, \textit{top}).\textit{Probe} \\ &\quad + (c, \textit{top}).\textit{Probe} \\ \textit{ProbeA} &\stackrel{\textit{def}}{=} \textit{ProbeB} + \textit{ProbeC} \\ &\quad + (a, \top).\textit{ProbeA} \\ \textit{ProbeB} &\stackrel{\textit{def}}{=} (b, \top).\textit{Probe} \\ \textit{ProbeC} &\stackrel{\textit{def}}{=} (c, \top).\textit{Probe} \end{aligned}$$

4.2 New Approach

The new approach uses a method borrowed from lexical analyser generators. To compile a probe its specification is first transformed into a non-deterministic finite state machine. This is then translated into a deterministic finite state machine. The deterministic finite automata is then minimised by combining equivalent states. Finally the self-loops are added and the resulting finite automata trivially translated to a PEPA component.

Minimising the dfa is not quite the same as a traditional dfa minimiser. Two states are considered equal if they have equal move sets. That is, two states s_1 and s_2 are considered equal if every action which can be taken by s_1 can be taken by s_2 and results in the same state and vice versa. For the purposes of the comparison any move that results in state s_1 or s_2 is considered the same (provided it has the same label). So for example if s_1 loops on action 'a' that is equivalent to s_2 looping on 'a' or performing 'a' and going to state s_1 . In fact if s_1 performs 'a' and goes to s_2 , and s_2 performs 'a' and goes to s_1 then the two moves are considered equal.

If two states are considered equal (because they have equal move sets) we delete one of the states, remove all moves which originate from that state, and change all moves which target that state to target the equivalent state. We then recursively minimise the dfa as this coalascing of states may lead to further comparable states. In the next section a full example of this sequence of compilation steps is reported, but first the compilation of labels must be reported.

4.3 Labels

In Section 2 labels were defined as indicating that significant events have occurred. These may be used as communication between multiple probes. To compile labels it must be ensured that the communication over which the probes communicate occurs faster than the model can perform activities. To achieve this, immediate actions are used. The PEPA language is extended to include immediate actions, these are high priority actions which are always performed before any timed activities that are currently enabled. A label is then compiled to an immediate action and multiple probe components can synchronise over the immediate communication signals in the normal manner.

The additional syntax $a.P$ denotes a prefix component which performs the immediate action a before becoming the component P . The probe $(a : start, b : stop)$ is therefore translated as:

$$\begin{aligned}
 Probe &\stackrel{def}{=} (a, \top).start.ProbeB \\
 &+ (b, \top).Probe \\
 ProbeB &\stackrel{def}{=} (b, \top).stop.Probe \\
 &+ (a, \top).ProbeB
 \end{aligned}$$

Notice that where the probe performs immediate communication there is no need to perform the self-loops since the immediate action will take priority over any timed activities that the model performs.

4.4 An Example

As an example consider the probe:

$$(a, b)?, c : \textit{start}, d : \textit{stop}$$

Figure 2 denotes the sequence of transformations done in `pepaprobe`. The graph on the very left hand side shows the probe translated to a non-deterministic finite automata. This then has the non-determinism removed from it to produce the deterministic finite automata shown in the graph second from the left. The third graph shows the deterministic finite automata minimised by combining equivalent states. Finally the self-loops are added to give the graph on the right hand side.

This probe is fairly simple so the minimisation manages to remove only one state. State 1 is removed because it is equivalent to state 0.

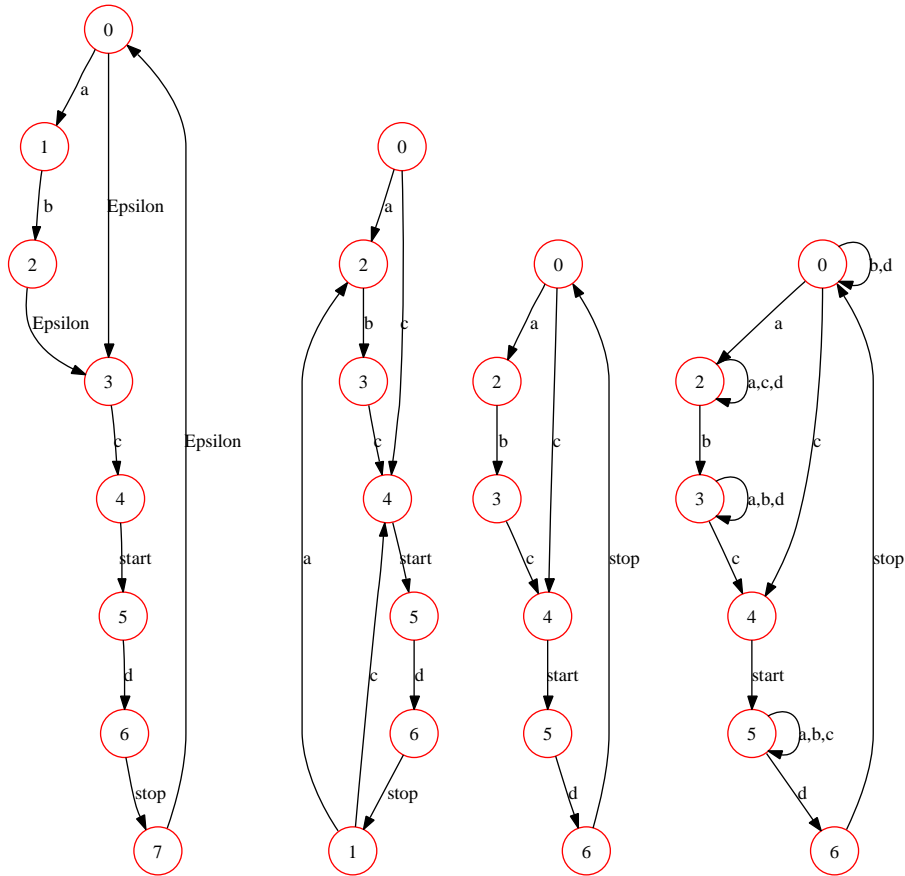


Figure 2: The probe on the very left hand side is the probe translated as an nfa. The second from the left shows the nfa translated into a dfa. The second from the right shows the dfa minimised. The final graph on the right depicts the final probe with the self-loops added.

5 Future Work

Both the probe language and the placement language may be extended. Ashok Argent-Katwala has proposed an alternative placement language though this is yet to be implemented in `ipclib`. The compilation using finite automata suggests that the user should be able to "create their own states". For many tasks the regular expression syntax with the implied states is more readable but there are some situations in which explicit states can aid readability. A simple example might be:

```
< ready > (break.broken | request : start), (break.broken | respond : stop)
< broken > (repair.ready | respond : stop)
```

This has two states *ready* and *broken* and describes a probe which measures between *request* and *respond* activities but does not start measuring if the *request* arrives when some component is broken. Note that a self-loop will be added in this state such that the probe can perform a *request* activity and therefore not disturb the behaviour of the model. However it will not start the measure in this case.

Finally the current implementation allows the user to specify whether the minimising of deterministic finite automata should occur before or after the addition of the self-loops. Testing using this option indicates that it should be safe to allow this. Since there are some cases in which this can result in a probe with fewer states I would like to prove that it is always safe and make this the default behaviour.

6 Acknowledgements

This work has been sponsored by the project SENSORIA, IST-2005-016004. Stephen Gilmore has participated in helpful discussions. Jeremy Bradley first created the `ipc` software tool – upon which `ipclib` and hence the `pepaprobe` utility are derived – and has since provided many helpful insights into its workings.

References

- [1] A. Clark, "The `ipclib` PEPA Library," in *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST)* (M. Harchol-Balter, M. Kwiatkowska, and M. Telek, eds.), IEEE, Sept. 2007. To appear.
- [2] A. Clark, "The `pepaprobe` manual." <http://homepages.inf.ed.ac.uk/s9810217/software/ipclib/web/ipclib.html>, 2007.
- [3] J. Hillston, *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
- [4] A. Argent-Katwala, J. Bradley, and N. Dingle, "Expressing performance requirements using regular expressions to specify stochastic probes over process algebra models," in *Proceedings of the Fourth International Workshop on Software and Performance*, (Redwood Shores, California, USA), pp. 49–58, ACM Press, Jan. 2004.

- [5] J. Bradley, N. Dingle, S. Gilmore, and W. 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* (G. Kotsis, ed.), (University of Central Florida), pp. 344–351, IEEE Computer Society Press, Oct. 2003.
- [6] J. Bradley, N. Dingle, S. Gilmore, and W. Knottenbelt, “Extracting passage times from PEPA models with the HYDRA tool: A case study,” in Jarvis [8], pp. 79–90.
- [7] M. Tribastone, “The PEPA Plug-in Project,” in *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST)* (M. Harchol-Balter, M. Kwiatkowska, and M. Telek, eds.), IEEE, Sept. 2007. To appear.
- [8] S. Jarvis, ed., *Proceedings of the Nineteenth UK Performance Engineering Workshop*, (University of Warwick), July 2003.